Formal RTL Sign-off with Abstract Models

Lucas Deutschmann*†, Osama Ayoub*§, Rohith Batthineni§, Michael Schwarz§,
Tobias Ludwig§, Dominik Stoffel†, Wolfgang Kunz†
†RPTU Kaiserslautern-Landau, Kaiserslautern, Germany §LUBIS EDA, Kaiserslautern, Germany
lucas.deutschmann@rptu.de – osama.ayoub@lubis-eda.com

***Abstract*—The complexity of today's hardware (HW) systems has exhausted the scalability of conventional register transfer level (RTL) design flows. The need for more efficient HW design and verification led to the introduction of abstract prototypes at the electronic system level (ESL). However, the semantic gap between such untimed ESL models and cycle-accurate RTL designs remains a critical issue, preventing HW sign-off at the higher abstraction layer. Existing approaches that aim to bridge this gap are often application-specific or do not establish a formal relationship between the two levels of abstraction. The concept of Path Predicate Abstraction (PPA) can establish formal *soundness* of the abstraction for general-purpose designs, however at the cost of high manual effort.**

In this work, we present three techniques to automate large parts of this abstraction technique and its corresponding design methodology. We propose a way to generate HW prototypes directly from the abstract model, taking advantage of the strengths of conventional code generation flows. Furthermore, we contribute two novel approaches to automate, or even obliterate, the most manually-intensive step of the original abstraction technique. We demonstrate the effectiveness of the proposed approaches in several case studies, including industrial designs.

I. Introduction

Abstraction has always been a key enabler for increasing scalability of digital hardware (HW) design. What began with grouping sets of transistors into standardized gates, continued with describing the behavior of the design with hardware description languages (HDLs) at the register transfer level (RTL). Modern synthesis tools support these abstraction layers seamlessly, relieving the HW designer from detailed knowledge about transistor-level and physical design. Still to date, RTL descriptions remain the golden reference model for HW sign-off in most applications [1].

However, the increasing complexity of today's System-on-Chip (SoC) designs exhausts the scalability limits of the RTL and demands an even higher level of abstraction. Nowadays, it is common practice to create abstract system models in software (SW) at the the so-called electronic system level (ESL). These models, often referred to as virtual prototypes (VPs), can vary in their degree of abstraction compared to the RTL descriptions which are always cycle-accurate. A widely used abstraction is transaction-level modeling (TLM). In TLM, the microarchitectural (bit- and cycle-accurate) details of transactions, such as the signal waveforms of a handshaking protocol, are omitted. Instead, TLM only considers the information transfer of transactions, i.e., what data goes from where to where. This abstraction allows for a significantly increased scalability, which is especially useful when simulating entire systems. However, the loss of cycle-accuracy makes it difficult to compare the behavior of a TLM model to the corresponding RTL design. This is known as the *semantic gap*.

The most prominent approach that aims to overcome this gap is high-level synthesis (HLS) [2]. In HLS, a cycle-accurate RTL description is generated from an abstract behavioral specification. Especially for data-centric accelerators, it produces efficient results and has been widely adopted in industry. However, HLS struggles to generate competitive designs for control-heavy systems [3]. Furthermore, there is no direct, explicit relationship between the abstract specification and the generated design, which means that potential changes at the RTL are very difficult to map back to the higher level. A different approach to closing the semantic gap, called *Path Predicate Abstraction (PPA)*, is proposed in [4]. PPA establishes a formal relationship between the different levels of abstraction so that any verification performed at the ESL is also valid at the RTL. PPA enables a sound top-down design methodology [5]. The approach can be facilitated with tool support, however, it still requires two manual steps: Creating the RTL design itself and refining the abstract model with its timing behavior. This manual effort can be infeasible, especially for complex systems or when there is a tight tape-out deadline.

In this work, we propose three novel approaches to simplify and increase automation of the PPA-based abstraction process between ESL and RTL beyond previous methods:

- *Operation-Level Synthesis (OLS)* (Sec. III) is a new approach to generating HW from abstract models. It incorporates conventional HLS, but complements its weaknesses when applied to control-heavy systems and establishes a formal relationship between the model and the generated RTL.
- *Operational Equivalence Checking* (Sec. IV-B) eliminates the detailed, internal refinement process between the RTL and the abstract model. Instead, end-to-end properties are used to verify that a manually created RTL design correctly implements the given PPA.
- *Automatic State Refinement* (Sec. IV-C) can be used to extract refinement information from a manually created RTL design. It follows the original methodology [5], but automates the refinement process to a large extent.

II. Background

A. Path Predicate Abstraction

Path Predicate Abstraction (PPA) [4] closes the semantic gap by constituting a formally *sound* abstraction between

*Both authors contributed equally to this research.

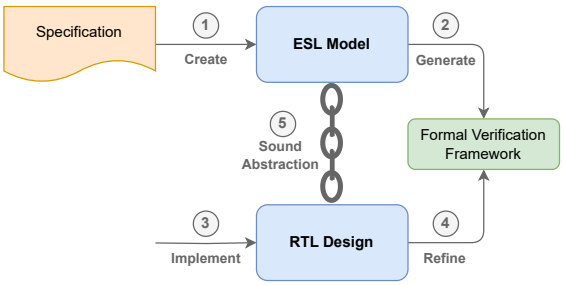

Fig. 1. Property-Driven Design Flow

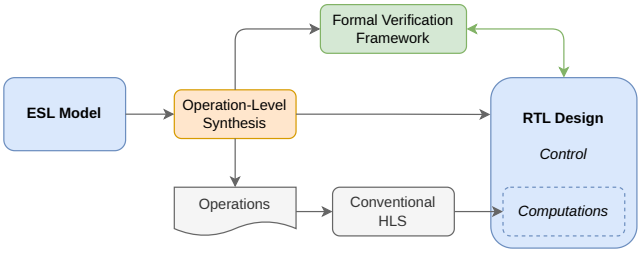

Fig. 2. Operation-Level Synthesis Flow

ESL and RTL. Through the establishment of a well-defined formal relationship, this technique allows untimed system-level properties to be mapped to a cycle-accurate description of the design. In particular, this means that any verification performed at the ESL is also valid at the RTL. We therefore believe that the PPA has the potential to contribute the required trust to sign-off intellectual property (IP) with untimed ESL models.

PPA models the behavior of the design as *operations* between abstract states. Operations can be viewed as single- or multiple-cycle transactions. In practice, abstract states are often represented by the main state machine of the RTL implementation responsible for the control behavior. Since microarchitectural details are omitted, such an abstract state can cover many concrete design states. In PPA, not every concrete RTL state needs to have a corresponding abstract state, as such an ”unimportant” state is implicitly traversed by an operation between abstract states. These characteristics of the PPA thus allow a single abstract model to have several possible concrete RTL implementations. Microarchitectural details are omitted in the abstraction, reducing the system-level model to mere functional behavior.

In order to ensure that a concrete RTL design implements a given PPA, a formal verification technique called Interval Property Checking (IPC) [4] is used. Every operation of the abstract model is translated into an interval property that starts and ends in an abstract PPA state. The way that these operations describe the entire behavior of the abstract model ensures that the set of properties is *complete* [4]. Each abstract state and each operation must then be refined towards the concrete RTL description and its timing characteristics. The verification of these properties on the RTL ensures that the design covers the same functional behavior as its abstract model. This formal relationship enables a top-down design methodology as in Sec. II-B. PPA can also be used in a bottom-up flow that raises the abstraction of existing RTL designs. This allows for creating sound ESL models of legacy designs. However, since the manual refinement step is identical for both approaches, we omit the bottom-up flow in this work.

B. Property-Driven Design

Property-Driven Design (PDD) [5] leverages the sound abstraction of the PPA to create a novel design methodology for HW. It takes inspiration from software engineering, where Test-Driven Development (TDD) [6] is a prominent programming paradigm. The main idea is that the verification framework, i.e., the formal properties, is created before and drives the design process.

Fig. 1 shows an overview of the the PDD flow. The method starts by creating and verifying an abstract system-level model, the PPA. From this model, a complete set of interval properties is generated. The design engineer then proceeds to create the cycle-accurate RTL description. For every functionality that has been implemented, the formal properties are refined with the timing information of the concrete implementation. When all properties have been refined, it is guaranteed that the RTL design has the same functional behavior as the ESL model. Consequently, all verification results verified at the abstract level also hold for the concrete implementation, creating a sound abstraction.

Two elements of the PDD methodology require high manual effort. Firstly, the RTL design itself is created manually. The second manual effort is spent refining the PPA with timing information of the RTL design. This process can quickly become cumbersome, especially if the level of abstraction between the ESL model and the concrete implementation is high. In this work, we propose several techniques to reduce this manual effort. We present a method that generates an RTL description from the PPA directly in a fully automatic fashion. Furthermore, we present new ways to minimize the refinement effort for manually created designs.

III. Operation-Level Synthesis

OLS automates the top-down design process of PDD. An overview of the approach is given in Fig. 2. OLS leverages the strengths of conventional HLS, combining it with the well-defined formal relationship that the PPA provides. HLS can efficiently generate data paths, but may struggle when applied to control-oriented systems [2], [3]. Therefore, in OLS, conventional HLS is only used to generate the compute-intensive parts of the system, while the control behavior is extracted directly from the abstract model. From the PPA model, all operations, i.e., the transitions between abstract states, are synthesized by a conventional HLS tool. The control logic, which keeps track of the abstract state and determines which operation to trigger next, is generated directly from the abstract model. Since the HLS tool is free to choose any timing, we employ a handshake protocol between the control logic and the computation module. In addition to the RTL design itself, a complete set of properties is generated and automatically refined.

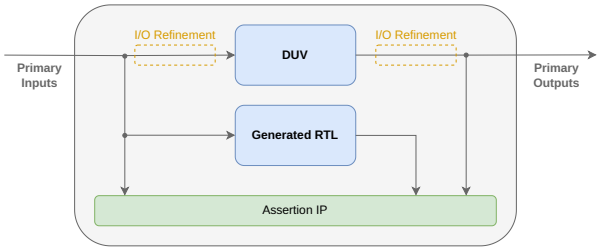

Fig. 3. Verification Wrapper setup

The OLS methodology can be used to quickly generate RTL designs that inherit the same functional behavior as their abstract ESL models. This allows the engineer to perform design space exploration or early integration testing. Furthermore, by utilizing advanced HLS tools, we can take advantage of all of their optimization features. For example, constraints can be applied to guide the synthesis process to split an operation into multiple cycles depending on power and performance requirements. However, the greatest benefit comes from the formal verification framework, which enforces the functional equivalence of the design with the abstract model. In particular, late changes or even a complete replacement of an OLS-generated prototype with a manually created design will not affect the functional correctness of the system, because the generated property set can be re-used any time to verify functional correctness and soundness of the PPA model.

IV. Verification Approaches

While OLS can be very useful for rapid prototyping, the performance and resource utilization of the generated designs are often inferior compared to manually created ones. Tight timing or resource constraints may require designs to be carefully crafted by hand using the conventional PDD flow (cf. Sec. II-B). In PDD, the engineer needs to manually refine the generated properties. However, this refinement process can be a non-trivial task that requires some expertise and knowledge of the RTL implementation. In this work, we contribute two approaches that can either automate or completely avoid the detailed refinement of internal signals.

A. General Setup

Fig. 3 shows the computational setup used for both approaches. The key idea is to use an OLS-generated design for reference when computing a refinement for the DUV. The analysis is performed operation by operation. For each operation, the reference design is used to constrain the DUV to the same abstract starting state. This is achieved by binding the primary inputs (PIs) of the design under verification (DUV) to the PIs of the generated RTL design and letting the formal tool inject equivalent input sequences triggering the PPA operation under consideration. Since the DUV may have a different timing behavior compared to the generated design, we still need to refine the interface with information about the timing of operations in the DUV. The output synchronization between the DUV and the generated RTL design is achieved by monitoring the validity signals of the output values. We store the output value of a specific operation in the generated design at the time point given by the OLS flow, typically one clock cycle after the operation was triggered. For the DUV, the user has to specify a latency range for the given operation.

This setup reduces the refinement effort of the abstraction to refining only the PIs and primary outputs (POs). For each operation, the setup constrains both the the DUV and the OLS reference design to the set of concrete states corresponding to the same abstract state. We can leverage this setup to perform operational equivalence checking (cf. Sec. IV-B) or to extract information about the implementation of the abstract state in the DUV (cf. Sec. IV-C). The corresponding formal properties of each approach can be generated directly from the abstract ESL model and are depicted as the assertion IP (AIP) in Fig. 3.

B. Operational Equivalence Checking

Operational equivalence checking compares an RTL design to an abstract PPA model. The setup of Fig. 3 has a similar structure as a classical equivalence checking miter. However, the two design instances may have different timing. If the DUV is functionally correct, both instances are refinements of the same PPA. Operational equivalence checking is performed operation by operation, by driving the DUV and the generated RTL design to the same abstract state, as described in Sec. IV-A, and then comparing their output behavior.

A suite of properties is generated that completely covers the functional behavior of the PPA. It contains one property for each combination of operation and PO. This is represented as the *Assertion IP* of the Verification Wrapper in Fig. 3.

```
1 property ec_property(op, i);
2   operation_sequence(op)
3   ##1 stored_val = OLS.PO[i]
4 |->
5   ##[0:delay(op)] stored_val == DUV.PO[i];
6 endproperty
```

Listing 1. Generalized equivalence checking property

Listing 1 shows a generalized equivalence checking property that is used to detect any difference between the output generated by the DUV and the output from the generated RTL. The presented property compares the value of output i for PPA operation *op*. The macro *operation_sequence*(*op*) triggers the given PPA operation in both, the generated RTL design and in the DUV. One clock cycle later, the output value of the generated RTL is stored. After a user-defined number of cycles *delay*(*op*) for the given operation, the output of the DUV is compared to the stored value.

Technically, the equivalence checking property can take several formats. One format accomodates for DUVs that are purely combinational. Another format handles DUVs that deliver outputs through a handshake mechanism, e.g., using *valid* and *ready* signals. In this case, we distinguish two types of properties:

- An *internal transition property* that is used to verify operations during which no valid output data is being produced. This means that only the validity of the handshake mechanism needs to be verified. The user-defined delay is used to specify the length of the operation in the DUV.

- A *communication point property* that covers operations whose endpoint is given by a handshake signal. The corresponding output of the DUV is checked against the generated RTL design once it is marked valid. The handshaking is verified in a separate property and requires to specify a latency range.

Operational equivalence checking can be leveraged to minimize the manual refinement effort by requiring the engineer only to consider the interface, thus overcoming the need to have knowledge of the internal structure of the RTL design. However, the complexity of its end-to-end style properties can become prohibitively expensive for systems with very long and complex operations. For such cases, we propose an alternative approach that aims at automatically extracting the internal state refinement.

C. Automatic State Refinement

As discussed in Sec. II, the conventional PDD approach requires a manual refinement step that maps information about the abstract model to the concrete RTL implementation. This step can be time-consuming because it may require detailed knowledge of the design's behavior and internal structure. In this work, we propose a new approach to extract the refinement of the abstract states in full automation. This approach uses the same setup as presented in Sec. IV-A and significantly decreases the manual time required for PDD.

The basic idea is to drive the DUV to each abstract state and determine the value of each signal in that particular abstract state. This process thus extracts information about the concrete RTL implementation and thereby determines the refinement of each abstract state. To generalize the notation, we denote S as the set of all bits in the design and define a *Refinement Vector (RV)* as the concatenation of a subset of S, i.e., $|RV| \leq |S|$. If signals of the design can be identified as not related to the HW control state, e.g., a cache or the register file of a processor, they can be omitted in the RV to reduce the complexity.

The proposed approach comprises three main steps:

1) Bit-Probing: For each transition between abstract states, each bit in the RV is checked with two formal properties to see if it evaluates to 0 (or 1). An example for such a property that probes the value 0 is given in Listing 2. If only one of the properties holds, we conclude that the abstract state requires that particular bit to be 0 (or 1). In case both properties fail, we cannot extract any information about the implementation of the abstract state.

```
property bp_property_0(op, i);
  operation_sequence(op)
|->
  ##1 RV[i] == 1'b0;
endproperty
```

Listing 2. Bit-Probing Property that checks if bit i evaluates to 0 after PPA operation *op* has executed.

This step results in a total of $2 * |RV| * |OP|$ bit-probing properties, where $|OP|$ is the number of operations in the abstract model. While this can produce in a large number of properties for large-scale designs, the complexity of each property is very low and they can be proven independently. Furthermore, this approach can be fully automated. In Sec. V, we discuss the scalability of the automatic state refinement in more detail when presenting our case studies.

2) Bit-Clustering: The bit-probing step might not be sufficient to create an unambiguous refinement. In some cases, information about certain bits of the DUV can only be extracted in combination with other bits. As an example, assume that for a given abstract state, some 2-bit signal of the DUV can evaluate to 00, 01 and 10, but never 11. In this case, bit-probing alone would not provide any information because each individual bit can take any arbitrary value.

```
property 2bc_property_00(op, i, j);
  operation_sequence(op)
|->
  ##1 RV[i] & RV[j] == 2'b00;
endproperty
```

Listing 3. 2-Bit-Clustering Property that checks if the concatenation of bits i and j can evaluate to 00 after PPA operation *op* has executed.

Therefore, in the second step, we perform an n-bit clustering of all remaining RV bits. We start with 2-bit clusters and increase their size until an unambiguous refinement is achieved. Listing 3 shows an exemplary 2-bit clustering property that checks whether it is possible that the two given bits evaluate to 00. These properties always fail, since bit-probing has already shown that there is no uniquely determined value for the given bits. For the bit-clustering properties, however, we are only interested in whether the given configuration is reachable or not. To this end, we let the formal tool compute a *witness*. If the property is unreachable, i.e., no witness exists, we can exclude the configuration from the state refinement.

For a cluster of k bits, we can generate up to $2^k * \binom{|RV'|}{k} * |OP|$ properties, where $RV' \subseteq RV$ denotes the bits in RV that were not identified during bit-probing. In our experiments, it proved sufficient to cluster only 2 bits at a time and to consider only adjacent bits, which kept the number of properties manageable.

3) Exclusion: In the final step, we combine the extracted information to create a refinement for each abstract state. An example is given in Listing 4. Since multiple operations can lead to the same abstract state, its refinement is the disjunction of the refinements extracted from each individual operation. Within each operation, the refinement consists of the conjunction of all bits extracted from bit-probing and the negation of all unreachable values from bit-clustering. Finally, to remove any overlap between state refinements, we exclude the refinement of all other abstract states.

```
// Abstract State 1 Refinement
(
  // Operation 1 Bit-Probing
  (RV[0] && RV[1] && RV[2] && !RV[3]) ||
  // Operation 2 Bit-Probing and Bit-Clustering
  (RV[2]) && !(RV[1] && RV[3])
) &&
// Exclusion
!(Abstract State 2 Refinement) &&
!(Abstract State 3 Refinement) && ...
```

Listing 4. Example of an extracted abstract state refinement

We performed the automatic state refinement on several

designs (for more details, see Sec. V). In each case study, the abstract states were correctly refined without any manual effort and without increasing the complexity of the PDD properties.

V. Experiments

We evaluate the approaches with case studies on several designs of different type and complexity. Tab. I gives an overview of the results. Runtimes are given in the format *hh:mm:ss*. We conducted all experiments using the commercial property checker OneSpin 360 DV (Version 2024.1_3) by Siemens EDA on an AMD EPYC 7502P 32-Core Processor with 256 GB of RAM running Ubuntu 22.04. All experiments are publicly available in our repository [7].

For each design, we measure:

- the number of abstract states, operations $|OP|$ and PI/PO in the PPA,
- the number of state-holding bits $|S|$ in the RTL design,
- the estimated manual effort for the internal state refinement required in the PDD [5] methodology, *this time reflects the manual engineering time saved by the approaches proposed in Sec. IV-B and Sec. IV-C,*
- the number of properties and cumulative property runtime if the operational equivalence checking is performed,
- the number of bits in the RV, the number of bit-probing $|BP|$ and bit-clustering $|BC|$ properties, as well as the cumulative property runtime.

A. Counter

This design implements a simple counter that counts up with a cycle delay when enabled and outputs the value via a dedicated PO. The PPA consists of one abstract "ready" state and four operations that cover reset, enable low, counting up and the overflow case. Due to the simplicity of the design, the runtime of the PDD and equivalence checking properties was negligible. The automatic state refinement successfully extracted the refinement of the abstract state in roughly a minute. The extracted refinement did not increase the complexity of the PDD properties compared to a manual refinement.

B. Arbiter

This design implements a round-robin arbiter. The PPA comprises a single abstract state and six operations. The runtime of the operational equivalence checking properties is similar to the PDD approach, while reducing the manual effort. Both approaches were able to detect a bug in the RTL implementation. In addition, the automatic state refinement was also successful and did not affect the runtime of PDD properties compared to a manual refinement. Since this IP is confidential, it cannot be published.

C. Processor

This design implements a sequential, unpipelined version of Hennessy and Patterson's DLX processor [8]. In a conventional PDD flow on this design, the manual refinement process requires more effort because the five abstract states are reflected in 16 registers at the RTL. The runtime of the operational equivalence checking properties was higher than for the conventional PDD flow, but with the advantage of not needing to manually refine the abstract states.

The automatic state refinement approach was successfully performed by generating 7880 bit-probing and 13852 bit-clustering properties. Despite the large number of properties, the extraction step only took about 10 minutes due to the efficiency of commercial model checkers. Additionally, no manual effort was required and each individual property finished instantly. In contrast, a manual refinement can take up to 4 hours depending on the experience of the verification engineer. The extracted refinement did not increase the computational complexity compared to a manual refinement.

D. Option Parser

The *Option Parser* takes a byte sequence of predefined length as input, parses it, and outputs a structure containing information about the sequence, including flags for special cases. This design has a complicated control flow because it processes the input stream sequentially, i.e., its state depends on the sequence of bytes already received, resulting in a high complexity proof for the model checker.

The runtime for the 152 operational equivalence checking properties is approximately 3 hours, which is significantly longer than for the PDD approach, in which running the properties took only 4 minutes and 30 seconds. However, the manual engineering time is much lower. A manual refinement of this module can take several days due to the complexity of capturing the abstract states. Several bugs, some of them corner-case bugs, were injected to demonstrate the complete coverage of the approach, even in highly complex designs. All were detected.

Nevertheless, the high runtime of the end-to-end-style proofs advocate a shift to the automatic state refinement, using the properties from the PDD approach. After a quick inspection of the design, 18 bits were considered control-relevant. Extracting the abstract state refinements took roughly 40 minutes due to the high complexity of the design, but did not increase the complexity of the PDD properties when using the refinement.

E. SHA512 Core

This module implements the SHA512 algorithm, which is commonly used to hash email addresses and passwords, and even plays an important role in blockchains.

The application of operational equivalence checking revealed the complexity of the end-to-end-style properties for proving the correctness of the message digest. Due to the large bit width of the data ports combined with the many rounds of the algorithm, this single proof did not converge or lead to a counterexample, even after running for several days. All other properties, such as verifying the correctness of the handshake, are proven immediately. This complexity issue is a general problem in formal verification. One possible solution is to break the property into several shorter steps. In the case of PPA, this would mean introducing additional abstract states.

The design contains many data registers, which would result in a large number of properties in automatic state refinement.

Design	PPA			RTL	PDD	Operational EC		Automatic State Refinement			
	States	$\|OP\|$	PI/PO	$\|S\|$	Est. Manual Effort	Properties	Runtime	$\|RV\|$	$\|BP\|$	$\|BC\|$	Runtime
Counter	1	4	1/1	9	$<$ 1 hour	4	00:00:05	10	80	72	00:00:12
Arbiter	1	6	2/1	224	$<$ 1 hour	6	00:00:06	291	3492	5972	00:02:32
Processor	5	20	2/4	156	2-4 hours	80	00:00:08	197	7880	13852	00:09:39
Option Parser	6	24	3/4	1630	2-3 days	152	03:07:56	18	864	320	00:37:54
SHA512	3	11	5/3	2067	1-2 days	15	–:–:–*	40	880	364	00:01:08

TABLE I
CASE STUDIES

However, these data signals were easy to identify due to their bit width, resulting in a refinement vector of only 40 out of 4330 possible bits. In our experiments, and we believe for the general case as well, the identification and exclusion of data path signals can be done without much effort. We will explore methods for an automatic exclusion in future work.Extracting the abstract states therefore took only 15 minutes to complete, and the extracted refinement did not increase the complexity of the PDD properties when used.

It should also be noted that the abstract model was successfully applied to an updated, more optimized version of the RTL design without much effort. This demonstrates the reusability of the abstract model for multiple design versions, boosting verification productivity in iterative design processes.

VI. Related Work

Several related methods for generating RTL designs from ESL descriptions have been proposed in the literature. HLS [2], [3] is by far the most prominent approach that can produce high-quality results, especially in data-centric accelerators. Other methods that aim to create universal generation frameworks generate RTL based on metamodels [9] or functional HW description languages [10]. However, in contrast to OLS (Sec. III), none of these approaches target a generation flow related to PPA [4]. PPA offers the unique advantage of not only allowing for a formally *sound* abstraction of individual modules, but also enabling a sound composition of multiple IPs into a system. As a result, system properties proven at the ESL hold "automatically" also on the composed RTL design, without further proof. In addition, OLS leverages the strengths of conventional HLS which is a mature and effective technique with wide adoption in industry.

There are several approaches that aim to prove equivalence between ESL and RTL using simulation [11] or formal methods [12], [13], [14]. However, none of these approaches use the notion of equivalent operations, nor do they employ a generated, cycle-accurate prototype to map the problem down to RTL–RTL equivalence checking. This allows us to leverage the power and flexibility of conventional and sophisticated model checkers combined with standardized languages like SystemVerilog Assertions (SVA). Nevertheless, we see some of these approaches and concepts, such as *event-based equivalence* [15], as complementary to our method and to PPA, and will explore possible synergies in future work.

VII. Conclusion

In this paper, we presented several ways to automate the sound abstraction process between ESL and RTL. With OLS, we introduced a method to generate IP prototypes directly from a PPA description. Leveraging these prototypes, we proposed operational equivalence checking and an automatic state refinement approach to overcome the complex manual refinement step of PDD. The efficacy of these methodologies was demonstrated through case studies on several designs. Our future work aims to further increase the robustness of the approaches and to apply them to more complex systems.

References

[1] M. Meredith and S. Svoboda, "The next IC design methodology transition is long overdue," *Open SystemC Initiative*, 2010.

[2] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.

[3] D. G. Bailey, "The advantages and limitations of high level synthesis for FPGA based image processing," in *Proceedings of the 9th International Conference on Distributed Smart Cameras*. ACM, 2015, p. 134–139.

[4] J. Urdahl, D. Stoffel, and W. Kunz, "Path predicate abstraction for sound system-level models of RT-level circuit designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 2, pp. 291–304, 2014.

[5] T. Ludwig, J. Urdahl, D. Stoffel, and W. Kunz, "Properties first—correct-by-construction RTL design in system-level design flows," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 3093–3106, 2020.

[6] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2022.

[7] "Formal RTL sign-off with abstract models," https://github.com/OsamaOAyoub/Formal-RTL-Sign-off-with-Abstract-Models.git, 2024.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[9] R. Kunzelmann, E. Baerens, D. Gerl, M. Bhadra, N. Schwarz, and W. Ecker, "A universal specification methodology for quality ensured, highly automated generation of design models," in *27th Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV)*. VDE/IEEE, 2024, pp. 90–98.

[10] F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, "Towards automatic hardware synthesis from formal specification to implementation," in *25th Asia and South Pacific Design Automation Conf.*, 2020, pp. 375–380.

[11] D. Große, M. Groß, U. Kühne, and R. Drechsler, "Simulation-based equivalence checking between SystemC models at different levels of abstraction," in *21st Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2011, p. 223–228.

[12] A. Koelbl, Y. Lu, and A. Mathur, "Embedded tutorial: Formal equivalence checking between system-level models and RTL," in *Int. Conf. on Computer-Aided Design (ICCAD)*. IEEE, 2005, pp. 965–971.

[13] C. I. C. Marquez, M. Strum, and W. J. Chau, "Formal equivalence checking between high-level and RTL hardware designs," in *14th Latin American Test Workshop (LATW)*. IEEE, 2013, pp. 1–6.

[14] J. Hu, T. Li, and S. Li, "Formal equivalence checking between SLM and RTL descriptions," in *28th IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2015, pp. 131–136.

[15] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva, "Towards equivalence checking between TLM and RTL models," in *5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2007, pp. 113–122.