Property-Driven Development of a RISC-V CPU

Tobias Ludwig[1], Michael Schwarz[1], Joakim Urdahl[1,2], Lucas Deutschmann[1], Salaheddin Hetalani[1], Dominik Stoffel[1], Wolfgang Kunz[1]

[1]Technische Universität Kaiserslautern, Kaiserslautern, Germany [2] now with Nordic Semiconductors, Norway
E-Mail: {*lastname*}@eit.uni-kl.de

***Abstract* — This paper demonstrates, at the example of implementing a RISC-V CPU, a novel top-down RTL design methodology called Property-Driven Development (PDD). The methodology starts from an abstract, transaction-level, hardware description in SystemC and produces a fully and formally verified RTL implementation. We present an open-source software tool supporting PDD. Using the tool, we develop an abstract system-level CPU model for the RISC-V R32I instruction set and refine it to an RTL implementation. Two different implementations for the same abstract model are provided.**

I. Introduction

This paper demonstrates a novel top-down RTL design methodology called *property-driven development* (PDD) at the example of implementing a RISC-V CPU. The methodology starts from an abstract, transaction-level, hardware description in SystemC and produces a fully and formally verified RTL implementation. To date, hardware models on the electronic system level (ESL) are extensively used in design exploration and virtual prototyping, but they rarely serve as a starting point and golden model for the actual implementation. As of today, the entry point for design implementation is still the RTL itself, simply because system-level models at the transaction level do not provide a clear and unambiguous "implementation view". This is due to the lack of a clear semantics. The PDD paradigm closes this semantic gap by establishing a well-defined and sound relationship between an abstract system-level model written in SystemC and the RTL. Basically, PDD ensures that the system-level model, at all times, remains a so-called *Path Predicate Abstraction* (PPA) [1] of the RTL implementation. In practice, this is guaranteed through the methodical use of formal property checking tools as they are commercially available.

PDD emancipates system-level models from prototypes to *golden design models* and enables moving complex verification tasks from the RTL to the system level where verification of global behaviors is usually much easier and less costly. In PDD, abstract properties are generated automatically in a standard property language from a system-level model. They are then refined together with the RTL implementation. At any stage during the design process the property suite can be verified against the RTL developed so far. This ensures that the RTL design implementation is a correct refinement of the system-level model, and, due to the formal soundness, also fulfills the same requirements and properties that the system-level model fulfills. The guidance of the RTL design flow by properties has additional advantages besides ensuring functional correctness. The properties generated from a well thought-out abstract model comprise valuable functional knowledge for the manual design process. As demonstrated by our case studies, this can point the designer to effective optimizations leading to more compact and more power-efficient implementations.

Sec. II provides an overview of the proposed methodology. Sec. III briefly covers the underlying theories. Sec. IV explains how PDD is related to other techniques. Practical aspects of PDD are elaborated in Sec. V to VII-D. Sec. VIII shows the results for the developed RISC-V CPU as well as for an industrial case study on a bus system.

II. Design Flow Overview

Fig. 1 shows the general flow of property-driven top-down design. It starts with a specification at the ESL formalized as an executable SystemC model. In order to establish a clear semantics with respect to the implementation, the SystemC model must be interpretable as a path predicate abstraction (PPA). Therefore, certain restrictions on the use of SystemC language constructs apply. We call this "PPA-designable subset" of the language *SystemC-PPA*. In SystemC-PPA, the modules to be implemented are represented as time-abstract, word-level instances of communicating finite state machines. Computation and datapath operations inside each module are described using standard SystemC operators. Communication between modules is modeled at the transaction level via event-based message passing. The SystemC-PPA model serves as both, an executable prototype at the transaction level, and a formal specification for the RTL design process.

The RTL design process splits up into two concurrent tracks. The RTL implementation track (left side of Fig. 1) is concerned with *how* the specified functions are implemented. The verification track (right side of Fig. 1) maintains the correctness of the implementation by verifying the property suite against the RTL model: it checks *what* behavior has been implemented. The starting points for both tracks, the RTL design and the property suite, are generated automatically from the SystemC-PPA description with the provided open-source software tool *DeSCAM* [2]. It generates an RTL skeleton in VHDL or Verilog that provides a framework for implementing the main controllers of each module. Furthermore, it generates a complete set of formal properties together with macros and functions that provide the user with means for mapping the system-level model to the implementation. During the design process, both, the RTL code and the property suite, are concurrently refined to include

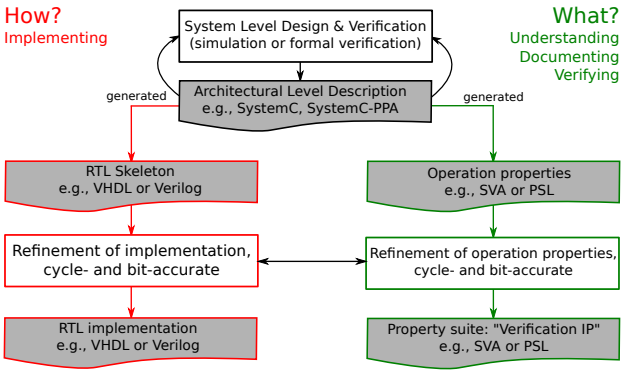

Fig. 1. Property-Driven Design — Work flow

details about the implementation, e.g., cycle-accurate timing and bit-accurate descriptions of the datapath, making use of the skeleton, macros and functions provided by the software tool.

When writing the RTL code the designer has full freedom over all aspects of the design, including the microarchitectural structure, the use of pipelining, the timing of individual operations, etc. The generated RTL skeleton only serves convenience purposes and may be replaced by other structures possibly preferred by the designer. What is important, however, is that the designer refines the macros and functions of the generated property suite. These macros provide the bit and timing information to relate the generated properties to the RTL. The properties themselves remain unchanged. As long as these properties can be formally proven on the RTL the design is correct in the strict formal sense. By construction, the final set of properties covers every behavior of the design. The resulting verification IP is therefore a valuable documentation and can be understood as a precise and formally proven data sheet of the design IP.

III. Theoretical Foundation

This section briefly covers the underlying theory of PDD. A detailed treatment of the theory can be found in [1], [3]. The goal here is to provide an intuitive understanding of PPA, as well as the meaning of the terms *soundness* and *operation properties*. In a design flow, there is an established chain of trust from the transistor level to the gate level, and from the gate level to the RTL. The correctness of a model at one abstraction level can be verified with respect to the next lower level. For example, an RTL model can be verified against its gate-level implementation by *formal equivalence checking* techniques such as combinational or sequential equivalence checking. Two sequential circuits are equivalent, if and only if they produce identical output sequences for all possible input sequences. Such a notion of equivalence does not exist between design models at the ESL and the RTL. At the ESL, a system is described by modules that communicate abstractly based on untimed message passing, whereas at the RTL, communication is specified with bit and clock-cycle accuracy.

For example, an ESL designer specifies reading a message from a bus as *bus→read*(). At the ESL this is represented event-based, for example, using handshaking. The RTL design implements the same operation with an arbitrarily complex, cycle-accurate bus protocol. The RTL design is considered sound w.r.t. the ESL if the protocol ensures, under all circumstances, that if the *bus→read*() is triggered a correct and valid message is transmitted from the writer to the reader. In this case, the RTL is a correct *refinement* of the ESL, and the ESL is a *sound* abstraction of the RTL. The term *soundness* describes a well defined formal relationship between an ESL and an RTL model. In the following we describe how this relationship is established. First, we give an intuition into the underlying theory of *path predicate abstraction* [1], and then we show how it is applied in practice in the context of RTL design.

A. Path Predicate Abstraction

Instead of revisiting the formal development of PPA from [1], we motivate its basic idea by considering a special graph labeling or coloring called “operational coloring”. Later, we show how this type of coloring is created using the concept of “operations” in digital circuits.

Definition 1 (Operational Graph Coloring): Consider a directed graph $G = (V, E)$, a subset $W \subseteq V$ of the graph nodes called *colored* nodes, a set of colors $\hat{W} = \{\hat{w}_1, \hat{w}_2, \ldots\}$ and a surjective coloring function $c : W \mapsto \hat{W}$. A path $(v_0, v_1, \ldots, v_n)$ such that $v_0, v_n \in W$ and $v_1, \ldots, v_{n-1} \in V \backslash W$ is called *operational path* in G. The set W must be chosen and colored such that:

1) every cyclic path in G contains at least one node from W (no cycles with only uncolored nodes in the graph),
2) for every operational path $(v_0, v_1, \ldots, v_n)$ and $u_0 \in W$ such that $c(u_0) = c(v_0)$ there must exist an operational path $(u_0, u_1, \ldots, u_m)$ in G with $c(u_m) = c(v_n)$

We call c an *operational coloring function* and G an *operationally colored* graph.

In other words, this definition considers a graph with two types of nodes, colored and uncolored. We call this coloring an operational coloring if the following two conditions are fulfilled: Every cycle in the graph must contain at least one colored node. A path starting in a colored node moving through uncolored nodes and ending in a colored node is called an "operational" path. The coloring must fulfill a second condition: If there exists an operational path that starts in some node with color 1 and ends in some node with color 2 then for any other node of color 1 there also must exist an operational path to color 2.

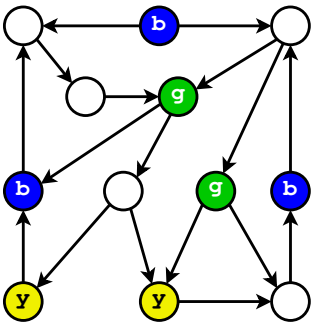

Fig. 2. Example of an operational coloring

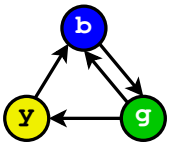

Fig. 3. Path predicate abstraction of Fig. 2

```
property my_operation is
assume:
    - - starting states
    at t+0:  a_start(S);
    - - trigger sequence
    at t+0:  a_0(X);
prove:
    - - output sequence
    at t+n:  c_1(Y);
    - - ending states
    at t+n:  c_end(S)
end property;
```

S: state variables, X: inputs, Y: outputs

Fig. 4. Example operation formalized as a property

Fig. 2 shows an example of an operationally colored graph according to Def. 1. The blue (b), green (g) and yellow (y) nodes are colored nodes, (they are elements of $W \subseteq V$). The nodes shown in white are considered uncolored, (they do not belong to W). As can be noted, in Fig. 2, every cycle in the graph has at least one colored node, so that the first condition of Def. 1 is fulfilled. Also the second condition is fulfilled. For example, from *every* green node there are operational paths to blue nodes as well as to yellow nodes, but there are no operational paths to any other color. Intuitively, the graph describes the "operations": green→yellow, green→blue, blue→green, yellow→blue. For the operationally colored graph of Fig. 2, we can draw an abstract graph as shown in Fig. 3. Such an abstraction is called *path predicate abstraction*.

Definition 2 (Path Predicate Abstraction): We consider a graph $G = (V, E)$ with a set of colored nodes $W \subseteq V$, a set of colors $\hat{W}$ and an operational coloring function $c : W \mapsto \hat{W}$.
A directed graph $\hat{G} = (\hat{W}, \hat{E})$, such that for any two nodes $u, w \in W$, it is $(c(u), c(w)) \in \hat{E}$ if and only if there is an operational path $(u, \ldots, w)$ in G, is called *path predicate abstraction* of G.

This definition states that the abstracted graph contains exactly one node for each color in the original graph, and path segments through uncolored nodes in the original graph are replaced by single edges in the abstract graph. Hence, every operation, such as blue→green, represented by several operational paths in the original graph is represented by a single edge in the abstract graph.

What is interesting about this abstraction? There is a well-defined formal relationship between the abstract graph and the original graph: Any path in the abstract graph can be described by a sequence of colors. Obviously, for any such path there is a corresponding path in the original graph that has the same sequence of colors, and vice versa. In this sense, path predicate abstraction is *sound w.r.t. color sequences*. As can be shown, soundness w.r.t. color sequences translates to soundness w.r.t. to certain properties formulated in the formal property language LTL (*linear time logic*). With respect to path predicate abstraction in digital circuits, it means that any relationship between certain events and operations on the ESL also holds in the implementation at the RTL (and vice versa).

In the following, we show how this relates to digital design. The graphs that we consider are the *state transition graphs* of finite state machines (FSMs), both on the RTL and on the ESL. An operational coloring of a state transition graph is a decomposition of the FSM behavior into operations that can be clearly defined using *operational properties*.

B. Operational Properties

The concept of an *operation* is as old as digital design itself. Intuitively, the term describes a piece of processing of digital data (computation or communication) that is executed in a finite number of steps, sometimes even within one clock cycle. For example, the execution of an instruction in a processor, or the the transactions in a communication protocol all constitute operations. The states at the beginning and at the end of an operation are *important* because they mark the "boundaries" to the previous or next operation, respectively. The intermediate states are inherent and specific to each operation and are therefore considered unimportant from an abstract point of view. Consider the state transition graph of the FSM implemented by a digital circuit at the RTL. Certain states can be marked as "important" in this state transition graph, others as "unimportant". Important states correspond to the colored nodes above, unimportant ones to the uncolored states. The paths between colored states are "operational paths" as in Def. 1 and can be unambiguously described by formal properties written in a standard property language.

Fig. 4 shows the general structure of an operation formalized in a property. (The property language shown is a pseudo-language. Any property language that allows to formulate LTL-type properties, such as *SystemVerilog Assertions* (SVA), can be used in practice.) An operation property starts in an important state (line 4) and has several trigger conditions, which are formulated on inputs or state variables (lines 5 to 6). If the operation is triggered, the property verifies that the correct outputs are generated (lines 8 to 9) and that the design ends in the desired state (line 11). The property specifies the concrete timing by referring to specific clock cycles in each line in the *assume* and *prove* part of the property.

For path predicate abstraction, operation properties are always formulated over a finite time interval (cf. Def. 1). They are called *interval properties*. The finiteness of the considered time window allows for efficient proofs based on *satisfiability (SAT) solving* (cf. [4]). This has enabled formal verification technology to handle industrial-size designs and led to powerful tools that are commercially available.

A set of operation properties can serve as a *formalized specification* of the design behavior, decomposed *functionally* in terms of operations. It complements the RTL model which is, mostly, a structural composition of blocks. However, the set of properties can only be viewed as a specification if it is *complete* [5], i.e., every possible design behavior is described by one of the properties in the set.

IV. Related Work

Modern industrial design flows incorporate an abstraction level above the RTL, called *Electronic System Level* (ESL). The ESL allows for design space exploration as well as early software development based on *Virtual Prototyping* [6]. SystemC has been very successful as an ESL modeling and simulation language, and has enabled new design and verification methodologies for abstract system-level designs [7], [8], [9], [10]. Due to the semantic gap between ESL and RTL, SystemC has so far not been used as the entry point for design implementation. Our methodology complements existing techniques, aiming to close the semantic gap and to upgrade system-level design as a new *golden reference* for subsequent design processes.

Note that PDD is only loosely related to *High-level Synthesis* (HLS). Usually, HLS is used in special domains like digital signal processing or other data-intensive applications. It is a largely automatic synthesis technique that can be complemented by *High-Level Equivalence Checking* (HLEC) [11], [12]. Also HLEC closes the semantic gap by exploiting a special notion of equivalence applicable to such designs and applications. HLS and HLEC are less suitable for describing and implementing the control and communication structures of a system. Our proposed methodology can therefore complement design flows in those cases where HLS and HLEQ are not applicable.

The PDD methodology is supported by an open-source tool called *DeSCAM* (described below). *DeSCAM* parses a SystemC description based on the LLVM/Clang framework [13] and uses parsing techniques, as introduced in [14]. After parsing the SystemC model, the framework generates an *Abstract Syntax Tree* (AST) that is used in a static analysis of the code, providing a semantically unambiguous representation of the model. The result of the analysis is a data structure describing the model and a report regarding SystemC-PPA compliance. Using Clang allows us to stay up-to-date with new C++ language features and to easily extend the supported language subset. Nevertheless, any EDA tool supporting the basic features of SystemC will be able to read in and process our SystemC-PPA models, including the SystemC frameworks [15], [16], [14]. There is related work [17] that derives properties from SystemC for implementation verification. The goal of these approaches is to maintain coherence between the test cases at different abstraction levels. In contrast, our work aims to establish a formal relationship between the system-level model and the RTL implementation.

A SystemC-PPA model is linked to the RTL by the verification track of Fig. 1. Our PPA-based methodology has been formulated in such a way that no knowledge of higher order logic or related languages is required to implement the proposed PDD paradigm. Only standard design and verification languages like SVA are needed. All proofs can be based on bounded circuit models using SAT-based property checking [18], [4], as it is commercially available.

V. Ingredients of Property-Driven Development

A PDD-based design flow consists of four major steps in transforming a SystemC-PPA model into a correct-by-construction RTL implementation. The first three steps are automated and happen inside our tool *Design from SystemC Abstract Model* (DeSCAM) [2]. The last step is a manual process. The four steps are:

1) Parsing and syntactic analysis of the SystemC-PPA model (cf. Sec. VI-A). In case the input model has syntax errors or is not SystemC-PPA-compliant, diagnostic feedback is provided to the designer. The result of this step is a data structure named *Abstract Model* (AM) consisting of structural and behavioral descriptions.
2) Extraction and optimization of the PPA (cf. Sec. VI-B). The PPA is the starting point for automatic property generation. This step optimizes the PPA representation in order to reduce complexity and enhance readability.
3) Generation of formal property suite (cf. Sec. VII-D). A set of abstract property descriptions is generated from the PPA in the desired property language.
4) Concurrent refinement of properties and RTL design. The generated properties are refined manually. This is interleaved with a conventional (manual) RTL design process.

The software *DeSCAM* is open-source and available on *GitHub*. The online repository includes an extended manual covering the SystemC-PPA subset, the examples and the plugin system that can be used to customize and extend the local design flow.

In the following sections we describe the Property-Driven Development flow in more detail. Sec. V-A discusses the basic structure and semantics of a SystemC-PPA system-level model. Sec. V-B highlights syntax and important modeling constructs in SystemC-PPA.

A. ESL Modeling With SystemC-PPA

In the following, we introduce the semantics of SystemC-PPA. The system-level model is executable, i.e., it can be simulated with any SystemC simulator. Communication between the modules in a system is modeled on the transaction level, i.e., the system behavior is given by time-abstract finite state machines described at word level. The FSMs send each other messages based on synchronization events. Each FSM is a PPA and describes one module. The FSM states represent communication events, and the transitions between states represent operations. The overall system behavior results from an *asynchronous product* [1] of the individual FSMs. This allows for a modeling of all interleavings of messages being passed between the modules, and ensures capturing all possible behaviors of the system. Due to the untimed behavior of the system-level model, each module is allowed to run at its own speed. In order to exchange a message between two modules, they need to synchronize through a handshake.

At system level this handshake is implemented through SystemC events. During the RTL design process the handshake is implemented explicitly as a four-phase "request-acknowledge" handshake. Sometimes, implementing full four-phase handshaking bears unnecessary overhead, e.g., when modules share a common clock, or in cases where losing a message is acceptable. SystemC-PPA, therefore, provides three different kinds of communication interfaces called *ports*. The three supported interfaces, in our experience, provide enough flexibility to support any communication scheme between digital hardware components. The type of communication interface is selected during the system-level design process.

Each interface generates a different kind of property suite and, therefore, affects the subsequent hardware design process. The basic interface is called *blocking* and implements a blocking message passing handshake. It ensures that a message is never lost. The *master/slave* interface is a variant of the blocking interface for synchronous communication. If it is known that one side (the *slave*) is always ready for communication then the other side (*master*) may communicate without waiting for synchronization. Finally, the *shared* interface models the behavior of a volatile memory.

In order to simulate and verify the system-level design, an executable description of the system is needed. The industry standard for executable system-level designs is SystemC, but the semantics of SystemC does not, *per se*, match the semantics of our formal model. SystemC, like many other high-level modeling languages employed in industry, is primarily a *software programming language* used to program a simulator for analyzing the behavior of the modeled systems. For example, SystemC is used by a framework of class hierarchies and macro definitions to describe the structure of hardware systems, with the associated behavior being modeled in C++. While C++ has clearly defined semantics as a programming language, the high-level objects defined in the SystemC class framework lack precise semantics with respect to the abstract hardware designs they are intended for. We solved this by restricting SystemC to a subset of certain constructs called SystemC-PPA.

The following example intends to provide an intuition into the semantics of a SystemC-PPA model for blocking message passing. Let's assume we are designing a CPU. Most designers will describe a CPU as a set of modules like ALU, register file, control unit, which are connected to each other via ports. For example, the *Control Unit* has an output port `next_instruction` and the *ALU* has an input port `next_instruction`. The *Control Unit* sends a new instruction, formalized as a message (e.g., `ADD rs1, rs2, rd`), to the *ALU*. If the *ALU* is still busy with a different instruction the *Control Unit* is blocked until the *ALU* is ready for the next instruction. The blocking message passing handshake is sometimes also referred to as *Rendezvous communication* [19]. (Rendezvous communication is an analogy: In order for two people to communicate they have to be at the same place at the same time. If either one is missing the other one has to wait.)

B. Overview of SystemC-PPA Syntax

We will illustrate some key features of SystemC-PPA using the example of a module definition in Fig. 5, however, without explaining every language feature used in the example. Details can be found in the online documentation to *DeSCAM* [2]. The module provides communication interfaces to other modules in the system. Lines 8 to 10 define three such interfaces. There is a *blocking* interface, corresponding to a full four-phase handshake between the communicating modules. When the example module accesses the interface (in line 17), the *read* function call blocks until the communicating module has called the corresponding *write*. When a blocking interface is refined to the RTL, the generated set of properties contains special synchronization functions/macros that need to be implemented by the RTL developer in order to guarantee a full four-phase handshake. In the provided RISC-V implementations this type of interface is used for the communication between core and memory. If the memory is slower than the processor the processor is blocked until it receives data from the memory.

When RTL designs are synchronized through a common clock, four-phase handshakes are not necessary. Line 9 shows an example of a so-called *master/slave* interface, avoiding such overhead in the RTL refinement. A master/slave interface refines to a *unilateral synchronization* scheme. It relies on the timing constraint that the *slave* module is guaranteed to be ready for communication whenever the *master* module sends a synchronization signal. The generated properties ensure that the timing constraint is met by the RTL design. The register file of a processor is a good example for using the master/slave interface. In the provided RISC-V implementation the processor and the register file share a common clock and the register file is a slave

```
1:  enum status_t {in_frame, oof_frame};
2:  struct msg_t { status_t status; int data; };
3:  SC_MODULE(Example) {
4:      SC_CTOR(Example):nextsection(idle)
5:              {SC_THREAD(fsm)};
6:      enum Sections{idle,frame_start,frame_data};
7:      Sections section, nextsection;
8:      blocking_in < msg_t > b_in;
9:      master_out < int >m_out;
10:     shared_out < bool > s_out;
11:     int cnt; bool ready; msg_t msg;
12:     void fsm(){
13:             while(true) {
14:                     section = nextsection;
15:     if (section == idle){
16:         s_out→set(false);
            //State idle_3
17:         b_in→read(msg);
18:         if (msg.status == in_frame){
19:                 s_out→set(true);
20:                 nextsection = frame_start;
21:                 cnt = 3;
22:         }
23:     }else if (section == frame_start){
            //State frame_start_2
24:         m_out→write(cnt);
25:         cnt = cnt-1;
26:         if (cnt == 0) {
27:                 cnt = 15;
28:                 nextsection = frame_data;
29:         }
30:     }else if (section == frame_data){
            //State frame_data_0
31:         ready = b_in→nb_read(msg);
32:         if (!ready) {
33:                 m_out→write(msg.data);
34:                 if (cnt == 0){nextsection = idle;}
35:                 cnt = cnt-1;
36:         }
37:     }
38: }}};
```

Fig. 5. SystemC-PPA example

to the processor. If the processor writes back a value, the register file is always ready and, thus, a four-phase handshake is not necessary.

An example of a third communication primitive, an *unsynchronized* port, is shown in line 10. It can be used to model volatile data (like sensor I/O) or to provide additional information together with some other communication port that is of the blocking or master/slave kind. The control and computation behavior of the module is described in the form of an FSM (cf. line 12 in the example). It is divided into *sections* that loosely correspond to operations of the design. However, the actual abstract states of the PPA defined by the module are given *implicitly* through the communication primitives *read*() and *write*(), as shown, e.g., in lines 17, 24 and 31.

The behavior described in *fsm*() is the following: Execution starts in section *idle*. Execution keeps repeating section *idle* as long as no new frame is detected. If a new frame is detected, the shared output is set to *true* and section *frame_start* is executed. Section *frame_start* is executed until *master_out* transmits the message 3, 2 and 1. Execution continues in *frame_data* until 15 frames have been received. After the 15th frame, execution continues in *idle*.

In general, the FSM structure of a SystemC-PPA module allows for modeling of arbitrary digital hardware. It has a sufficiently high level of abstraction so that a human designer can quickly grasp the intended behavior. Not all details of the language can be discussed here, but from the shown example the reader should have a basic intuition about SystemC-PPA in order to understand the following sections.

VI. Tool Support for PDD

Once a SystemC-PPA model has been created, the user runs the automatic tool *DeSCAM* in order to generate both, properties for verification and an RTL design skeleton. In this section we describe the internals of this process consisting of several steps.

A. Abstract Model Generation

The first step consists of parsing the input file and generating an abstract data structure for further analysis. For parsing, we use the open-source compiler framework LLVM/Clang. The parser produces a representation of the SystemC-PPA input as *abstract syntax tree* (AST), containing the program code and all static information related to the program, like, e.g., the SystemC scheduler. The AST implements a software design pattern called visitor pattern, enabling an efficient subsequent analysis.

In the next step, the AST is analyzed and all information required for property generation is extracted. All details that are relevant only for C++ analysis or SystemC simulation, like the SystemC scheduler, are stripped away. The remaining information describing the module in a PPA view is stored in a new data structure called *Abstract Model* (AM). It contains structural information in the form of an abstract syntax tree and behavioral information as a control flow graph (CFG). Initially, a CFG is automatically generated by Clang for the class method *void fsm()* of a SystemC-PPA module. This method needs to follow a certain structure such that (1) its behavior, when simulated by the SystemC framework, is that of a finite state machine (FSM), and (2), that the FSM in terms of states and transitions can be extracted from the control flow graph of the SystemC code.

In fact, this FSM extraction is straightforward. The CFG nodes can serve directly as nodes of the state transition graph of the FSM, with a few exceptions, such as the outer *while (true)* loop, the *if-then-else (ITE)* structure for the sections and any *nextsection* assignment (if present). *DeSCAM* removes the CFG nodes not needed and produces a CFG that represents the FSM of the PPA, as shown in Fig. 6. The nodes are labeled with the line numbers of the statements in Fig. 5. The dashed boxes indicate the sections the individual CFG nodes belong to. The tool analyzes each object of the SystemC/C++ program and checks for compliance with SystemC-PPA. Violations are reported in form of warnings and errors and the affected objects

are rejected. Note that, often, ESL models written in SystemC serve several purposes at the same time, e.g., to support early firmware development, to help in integration testing and to generate tests. SystemC models may therefore contain code that is meaningless in the PPA view but should still remain in the source. When *DeSCAM* warns about code that is not SystemC-PPA compliant the user needs to analyze the diagnostic output and make a decision on whether the affected code should stay included in the model or not.

B. PPA generation

In the next step *DeSCAM* generates a PPA from the Abstract Model. This step has three phases:

- Coloring: Operational coloring is applied on the initially uncolored CFG. In case of PDD the coloring is trivial, because the important states are implicitly provided by the communication primitives. Each communication with synchronization results in an important state.
- Identification of operational paths: Every path between two important states (cf. Sec. III) in the CFG results in an operation and will result in an operational property.
- Optimization: The PPA is optimized with respect to structural complexity, in order to enhance conciseness and readability.

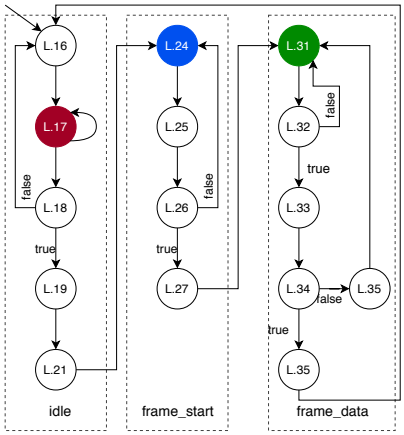

Fig. 6. Generated uncolored CFG

```
1:  property red-to-blue is
2:  assume:
3:      - - starting states
4:      at t+0:  state == L. 17;
5:      - - trigger sequence
6:      at t+0:  sync == true;
7:      at t+0:  msg.status == in_frame;
8:  prove:
9:      - - output sequence
10:     at t+n:  s_out == true)
11:     at t+n:  cnt == 3;
12:     - - ending states
13:     at t+n:  state == L. 24
14: end property;
```

Fig. 8. Operation property: red to blue

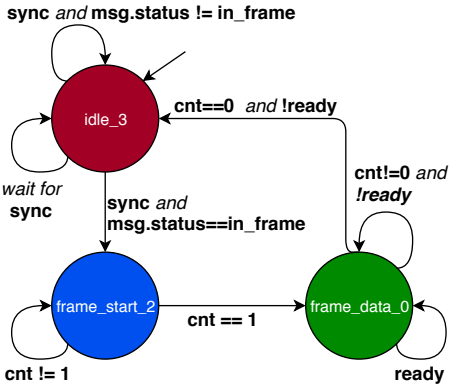

Fig. 7. Resulting PPA

Prior to applying the operational coloring, the CFG is extended by additional operations that are defined implicitly by the SystemC-PPA model, but are not explicitly represented in the CFG:

- **reset operation**: It is defined by the initialization of the `nextstate` variable in the constructor (`SC_CTOR`) of the `SC_MODULE`, cf. Fig. 5. In the current version, *DeSCAM* supports SystemC-PPA modules with a single constructor only. The reset operation ensures the correct initialization of RTL registers and guarantees that the design is in the correct important state after reset.
- **wait operation**: Wait operations are generated for communications over *blocking ports*. If *read*() or *write*() is called and the counterpart of the communication is not ready then the module blocks, i.e., needs to wait. A wait operation is added to the PPA that forces the module to remain in its state until the counterpart is ready for communication.

After the insertion of implicit operations, operational coloring as required for generating the PPA is applied. First, every node resulting from a blocking communication is colored (i.e., red and green). A communication primitive implementing the *master* interface is colored only if condition 1 of Def. 1 is violated, i.e., a cyclic path in the CFG must be broken. This is the case for line 24, thus *L. 24* is colored blue. The communication primitive at line 33 does not violate the condition because there is no path without a colored node starting from *L. 33*. The paths end in either green or red.

Fig. 8 shows one of the resulting operation properties. This property specifies the transition from red to blue. It verifies that after a successful handshake and detecting a new frame in control state *L. 17*, the operation will always end up in control state *L. 24*, with *counter* set to 3 and with the shared port set to *true*.

In contrast to Fig. 6, *DeSCAM* names the states not by the line of code in which they are declared. Instead, as Fig. 7 shows, states are named by the section they are declared in, extended by a unique ID generated by *DeSCAM*. If multiple communication calls occur within one section they are distinguished by the unique ID. In our example, *L. 17* is renamed to *idle_3*, *L. 24* to *frame_start_2* and *L. 33* to *frame_data_0*.

The operations of the PPA are labeled with the trigger conditions. The effects of different port interfaces on the RTL implementation become apparent in Fig. 7. The state *idle* has a wait operation because the communication counterpart may not be ready when the module enters this state. On the other hand, the state *frame_data_0* does not have a wait operation

because the communication call uses the non-blocking *nb_read()* method. The state *frame_start_2* requires no handshaking, because the respective port implements the *master* interface.

Initially, each variable specified at the system level leads to the implementation of corresponding registers in the RTL. If a variable is only a temporary one storing intermediate computation results, an RTL register is not needed. In the given example, this is the case for the variable *ready*. The variable is assigned the status value indicating success of communication via port *b_in*. It is used only once in the if-then-else at line 32 where it can be safely replaced by the status value itself, in all operations containing line 32. By contrast, the variable *cnt* at line 35 is assigned its previous value, decreased by one. The new value depends on the previous one and thus the variable is a necessary part of the state space of the PPA resulting in an RTL register.

VII. RTL Implementation and Refinement

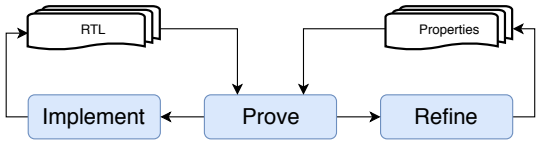

Fig. 9. Implementation and refinement

The final, manual, step of PDD is the actual implementation of the RTL from a PPA. Its workflow is depicted in Fig. 9. The starting point is the complete set of properties as *DeSCAM* has generated it from the PPA, where each property represents one operation that needs to be implemented at the RTL. For each abstract object of the system level like variables, data values, communication function calls using one of the primitives, etc., a *macro* or *function* definition (in SVA) is generated. The operation properties are built using these macros. For example, the macro *cnt* in Fig. 8 is one resulting from the system-level variable with the same name, *cnt*.

The designer implements the RTL code matching the properties, one property at a time. Each property is refined concurrently with the implementation, i.e., timing information is added as well as detail concerning the datapath implementation. If the chosen property holds on the design, the hardware implements the operation correctly, otherwise the RTL code needs to be corrected (or the timing adjusted). Once all properties can be verified successfully on the design then the design process is finished. The property suite provides a valuable verification IP for the finished design. It can be used as a formalized data sheet which documents precisely what functionality has been implemented.

Sec. VII-A explains in more detail what a macro is and how it relates an abstract object of the system level to a concrete RTL implementation.

A. Macros

Macros are an important vehicle for abstraction. By encapsulating references to RTL objects at different time points they decouple RTL implementation details from their abstract representation in a property. The property definitions themselves remain unchanged during the design process. Only the macro bodies are edited. This approach has two benefits. Since the generated set of properties is complete, independently from the actual content of the macros, it is not possible to introduce a verification gap during the refinement process; also, instead of the whole property text only a small part of the generated lines of code have to be adapted by the designer. We evaluated the associated work effort. Results can be found in Sec. VIII.

In this and the following section we illustrate the abstraction constructs using the commercial property language *ITL* [20]. Note that *DeSCAM* also supports SVA printout for which it generates *defines* and *functions* instead of *macros*. The interested reader can find the RTL implementation as well as the refined properties (both in ITL and SVA) for the example of Fig. 5 on our GitHub site.

Fig. 10 shows an abstract definition of a macro, consisting of a name, return type and a body as it is initially generated by *DeSCAM*. Within the *macro body* the designer specifies the implementation details of the abstract object. Sometimes, a word-level RTL variable has the same name and an equivalent type as the system-level variable it implements. For example, Fig. 11 shows such a straightforward refinement for the abstract system-level variable *cnt*. A variable can also be mapped to a more complex RTL data structure, as shown in the example of Fig. 12 where "cnt" is represented by a composition of two slices named "foo" and "bar".

```
1: macro MACRO-NAME: RETURN-TYPE :=
2:     MACRO BODY
3: end macro
```

Fig. 10. Macro definition

```
1: macro cnt: int :=
2:     instance/RT_signal
3: end macro
```

Fig. 11. Simple refinement for *cnt*

```
1: macro cnt: int :=
2:     instance/foo[31 downto 16]
3:     & instance/bar[15 downto 0]
4: end macro
```

Fig. 12. Non-trivial refinement for *cnt*

For refining the macros there are only three rules:

- Sequences may be formulated only over finite time windows (of arbitrary length).
- Functions characterizing abstract inputs may be expressions over only input signals of the RTL design.
- Functions characterizing abstract outputs may be expressions over only output signals of the RTL design.

Ports are modeled by a combination of three specialized macros: *notify*, *sync* and *datapath*. The combination of these macros is determined by the type of communication interface the port implements. A port with a *shared* interface requires no synchronization, because it models a volatile access. Hence, only a *datapath* macro is generated for it. The datapath macro describes the message and is named *portname_sig*. The operation properties guarantee that the correct message is sent by specifying a value for the datapath macro. Ports with a *master/slave* interface require, generally, no synchronization. The output port with a master interface is complemented by *notify* indicating validity of a new message for the respective slave input. Conversely, the slave input has a *sync* macro in order to evaluate the validity of the incoming message. The macros are named *portname_notify* and *portname_sync*, respectively.

For a port implementing a *blocking* interface both macros, *notify* and *sync*, are required to implement a handshaking mechanism. The four-phase handshake starts with raising the outgoing *notify* flag. The module is blocked until the incoming *sync* flag evaluates to *true*, indicating readiness of the counterpart. At the end of the transmission both flags evaluate to *false*. The correct handshaking is enforced by the generated operation properties. The design has to fulfill these properties, resulting in a correct-by-construction handshaking. The evaluation of the macros is not necessarily restricted to a single signal changing its value in a single clock cycle. The macros may describe an arbitrary protocol spanning multiple cycles and different signals.

Macros for datapath registers result from variables at the system level. The macros for compound types are split into separate macros for each subtype. For example, the variable *msg* is separated into two macros *msg_data* and *msg_status*. The same idea applies for port macros. The provided example implementation has three datapath registers: one for the variable *cnt* and two for the variable *msg*, namely *msg_data* and *msg_status*. As explained earlier, the variable *ready* is not required for the RTL implementation.

Important states, derived from the communication calls at the system level, each result in a macro. Fig. 7 shows the resulting PPA with three important states, *idle_3*, *frame_start_2* and *frame_data_0*. Each important state results in a macro of *boolean* return type. If the hardware is in an important state the macro evaluates to *true*, otherwise to *false*.

B. RTL Skeleton

As a starting point, the PDD methodology requires, at the least, a minimal RTL description that provides structural information about the implementation, such as the ports and the needed datapath registers. We call it *skeleton* in the sequel. A behavioral description is not needed for the first iteration of PDD. The designer can either use the skeleton provided by *DeSCAM* or create a custom one. It is even possible to start with a pre-existing design and then only edit the properties for refinement. This is a promising solution for dealing with legacy design code, as will be shown in Sec. VIII-B.

In the following, we show how to create an RTL implementation for the PPA example of Fig. 5. We begin with the skeleton as it is generated by *DeSCAM* and show how the corresponding properties are refined. We explain refinement of macro definitions at the example of the macro generated for port *b_in*. We show how to implement RTL code at the example of the reset operation. Fig. 13 shows the RTL skeleton generated by *DeSCAM*. A VHDL package declaring the needed data types is generated together with the skeleton. The code is somewhat simplified for demonstration purposes. The full example is available on GitHub.

```
 1: entity Example is
 2: port(
 3:     clk: in std_logic;
 4:     rst: in std_logic;
 5:     - - b_in: in msg_t;
 6:     b_in_sync: in bool;
 7:     b_in_notify: out bool;
 8:     data_in: in signed(0 downto 0);
 9:     m_out: out int;
10:     m_out_notify: out bool;
11:     s_out: out bool )
12: end Example;
13:
14: architecture Example_arch of Example is
15:     signal section: Example_SECTIONS;
16:     signal cnt_signal:int;
17:     signal msg_signal:msg_t;
18: begin
19: control: process(clk)
20:     if (clk='1' and clk'event) then
21:             if rst = '1' then
22:                     section <=idle;
23:                     cnt_signal <= 0;
24:                     s_out <= false;
25:                     b_in_notify <= true;
26:                     m_out_notify <=false;
27:                     msg_signal.data <= to_signed(0,32);
28:                     msg_signal.status <= in_frame;
29:             else
30:                     - - Implement the control behavior
31:             end if;
32:     end if;
33: end process
34:
35: end Example_arch;
```

Fig. 13. RTL implementation

The skeleton declares a port *b_in* (line 5) that transports a message of type *msg_t*. It consists of a 32-bit integer for *msg.data* and a 1-bit *boolean* for *msg.status* (cf. Fig. 5). In practice, a system-level data object can usually not be simply "copied" to

the RTL, but needs to be refined to a low-level representation according to specific data formats and communication protocols. In our example, the 33-bit wide input port data, as generated by *DeSCAM*, is represented on the RTL as a serial bit stream received in 33 beats over a 1 bit-wide input. In Fig. 13 a 1-bit input port called *data_in* (line 8) is added and *b_in* (line 5) is removed. The protocol is abstracted at the system level. The untimed word-level exchange of a message at the system level is transformed into a cycle- and bit-accurate exchange. *DeSCAM* generates two macros for the port *b_in*: *b_in_sig_status* and *b_in_sig_data*. In the following, we describe how these macros are refined for this specific protocol.

C. Refinement

Fig. 14, illustrates how the macro for the datapath *b_in_sig_data* is refined in order to implement the protocol. The property checker proves the property for an arbitrary starting state (resembled by arbitrary time point *t*). The other time points referred to in the property are finite offsets from *t*, i.e., the property covers a finite time interval of behavior. The first bit is received at timepoint *t*, the last bit at timepoint *t+31*. The macro describes the reception of this serially transmitted message. The method *prev(signal, n)* provides the value of *signal n* cycles prior to *t* and the method *next(signal, n)* returns the signal value at *n* cycles after *t*. Lines 2–3 describe the sequential behavior using the *next* function. The protocol for the refinement of *b_in_sig_status* is the following:

- The abstract bit is evaluated over the last four input bits of *data_in*.
- If the sequence is equal to *'1111'*, then the status bit evaluates to *in_frame* and otherwise to *oof_frame*.

```
1: macro b_in_sig_data : int :=
2:     data_in & next(data_in,1) & next(data_in,2) & next(data_in,3)
3:     & next(data_in,4) & ... & next(data_in,31)
4: end macro
```

Fig. 14. Refinement of macro *b_in_sig_data*

```
1: macro b_in_sig_status: int :=
2:             frame_detected( prev(data_in,4) & prev(data_in,3)
3:             & prev(data_in,2) & prev(data_in,1) )
3: end macro
```

Fig. 15. Refinement of macro *b_in_sig_status*

Fig.15 shows the resulting refinement and Fig. 14 describes a 4-bit integer composed of the value of the last four cycles. In this case, the RTL evaluates four bits to determine the status, whereas at the system level only one bit is required. The helper function *frame_detected* evaluates this integer and returns the required value.

Refining the important states boils down to specifying which state bits of the global state vector describe the important states. In general, the designer is free to describe the important states to his/her convenience.

D. Implementation

The first property a design must implement is the reset property. Fig. 16 shows the reset property of our example. Line 3 calls a macro named *reset_sequence*. As the name suggests, the macro defines the sequence of signal values that reset the circuit. In the simplest case, the macro describes an assertion of a signal called `reset` or the like. (The macro *reset_sequence* is not part of the PDD methodology. We use it here only to hide the details of circuit initialization.) The property specifies that when the reset is triggered the design has to fulfill the commitments in lines 5 to 14. The datapath registers must be initialized correctly (lines 6 to 8). The output *s_out* is required to become *false*. As mentioned in Sec. VII-A the properties ensure a correct handshaking. The reset property proves that after reset the hardware is in state *idle_3*. This state has been generated for the communication through port *b_in*. The read from this port is initiated by asserting its *notify* signal. The port *m_out* is not used and thus *m_out_notify* evaluates to *false*. Fig. 13 contains the implementation of the reset operation between lines 22 and 28. Now, after the reset operation has been implemented, the PDD process continues with the operations starting in important state *idle_3*.

In our example, we continue with the operation leading from *idle_3* to *frame_start_2*, as described in lines 17 to 24 in Fig. 5. The corresponding operation property is shown in Fig. 17. There are two new undiscussed ITL language features in this property (line 2 and line 4). The first feature, `for timepoints`, is used to define the length of an operation by providing a value for the timepoint *t_end* at line 3. Note that the minimum length of an operation is one clock cycle. In case *t_end* is not defined, *DeSCAM* generates the property with the default value *t+1*. The property specifies that the operation receives a message at port *b_in* and modifies the datapath registers accordingly. The timepoint *t_end* of macro *msg.data* is changed to *t+32*. The operation has a length of 32 cycles, because it takes 32 cycles to receive the 32 serial bits. For example, the evaluation of macro *b_in_sig_data* at *t_end* results in the values from *t_end* to *t_end+32*.

The second ITL language feature, `freeze` (line 4), allows to associate the value of an expression at a specific timepoint *t* with a name so that it can be referenced later. For example, the "freeze variable" *b_in_sig_data_at_t* is assigned the evaluation of macro *b_in_sig_data* (the value of the received message data) at timepoint *t*. The property verifies (line 18) that the datapath register *msg_data* stores the message, received at timepoint *t*, at timepoint *t_end*, correctly. The property ensures the correct handshaking by checking that the associated *notify* flags only change value at the end of the operation, proven by line 21 and line 22. The important state entered after receiving the message is *frame_start_2*. This state results from the port *m_out*. The macro of this port has to evaluate to *3* (line 17) and the according *notify* is set (line 23).

```
property reset is
assume:
    reset_sequence;
prove:
    - - output sequence
    at t:     cnt = 0;
    at t:     msg_data = 0;
    at t:     msg_status = in_frame;
    at t:     s_out_sig = false;
    at t:     m_out_notify = false;
    at t:     b_in_notify = true;
    - - ending states
    at t:     idle_3;
end property;
```

Fig. 16. Reset operation

```
property idle_3_read_5 is
for timepoints:
    t_end = t+32;
freeze:
    b_in_sig_data_at_t=b_in_sig_data@t,
    b_in_sig_status_at_t=b_in_sig_status@t;
assume:
    - - starting states
    at t:     idle_3;
    - - trigger sequence
    at t:     b_in_sig_status=in_frame;
    at t:     b_in_sync;
    at t:     cnt==0;
prove:
    - - output sequence
    t_end: cnt = 3;
    t_end: m_out_sig = 3;
    t_end: msg_data=b_in_sig_data_at_t;
    t_end: msg_status=b_in_sig_status_at_t;
    t_end: s_out_sig = true;
    during[t+1, t_end]: b_in_notify=false;
    during[t+1, t_end-1]: m_out_notify=false;
    t_end: m_out_notify = true;
    - - ending states
    t_end: frame_start_2;
end property;
```

Fig. 17. Regular operation

The designer is entirely free on how to implement this operation. The provided example on GitHub is one possible implementation. Theoretically, there is an infinite number of sound refinements for the same PPA. If all properties hold on the design it is guaranteed with mathematical rigor that the implementation is sound w.r.t. the PPA and thereby sound w.r.t. the SystemC-PPA. It is, however, important that the designer is only allowed to change the length of the operation and the bodies of the macros. The property descriptions calling the macros must remain as generated by *DeSCAM*.

VIII. Experimental results

The proposed methodology has been evaluated by means of two case studies. The first study provides three different RISC-V implementations derived from the same SystemC-PPA. The goal of this study is to evaluate the effort for refining the properties to different designs, the effort for proving the properties on the design and the amount of abstraction between the system level and the RTL. The second study, an industrial case study, demonstrates that PDD is applicable to industrial-scale designs, leads to cleaner code and enables aggressive optimizations w.r.t. non-functional design goals.

A. RISC-V

The first case study comprises multiple implementations of a RISC-V CPU, each being a sound refinement of the same system-level model. The system-level model is a SystemC-PPA-compliant *Instruction Set Simulator* (ISS) implementing the *RV32I Base Integer Instruction Set*, as specified in [21], excluding interrupts. In the sequel we refer to the SystemC-PPA model as ISS. The methodology is elaborated on three different implementations:

- Simple sequential CPU. The design is implemented with two modules. The core consists of a CPU module and a register file. Datapath computations are described mostly by combinational functions. The goal of this implementation is to provide an RTL design requiring as little refinement effort as possible.
- Complex sequential CPU. The functionality of the processor is divided into four modules. The core is composed of a decoder, an arithmetic logic unit (ALU), a register file and a control unit. All communication between the modules is realized by *master/slave* interfaces. This CPU demonstrates how a complex communication structure implemented at the RTL is abstracted at the system level.
- Pipelined CPU: The CPU consists of a control unit, a data path and a register file. The processor is implemented as a five-stage static pipeline with forwarding. The control unit takes care of the pipelining and sets the respective control signals. The data path implements the computation and communicates with the register file.

The complete set of properties was generated for the ISS and refined for each implementation, resulting in three different property suites. Fig. 18 shows the PPA of the ISS. The ISS is connected to a memory by a *blocking* output port for sending a new memory request and a *blocking* input port for receiving the response. As we explained in Sec.VI-B, each blocking communication results in an important state. If the handshake fails, the system waits in its current state, as symbolized by the wait-operation of each important state. An instruction cycle starts in *request instr*, requesting an instruction from the memory. The ISS transitions to state *receive instr*, if the request is successful. Upon receiving the instruction the execution continues,

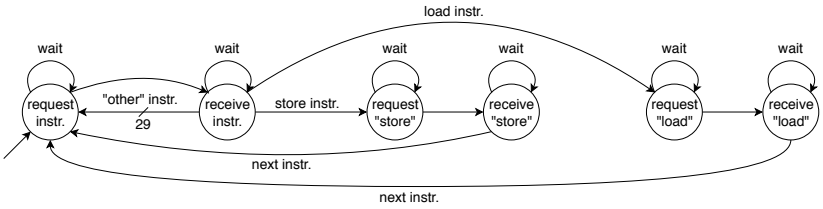

Fig. 18. PPA of the ISS

depending on the decoding of the instruction. Two cases need to be distinguished: In case a load or store instruction is received the ISS needs to make another communication to memory, resulting in state *request load* for a load instruction and *request store* for a store instruction. The state *request load (store)* is followed by another state *receive load (store)* reading the response from the memory. Execution then continues with fetching the next instruction from the memory. In case of any other type of instruction (e.g., R-type, I-type or B-type) the operation "other instructions", subsuming the operations for each of those encode types, is triggered. Basically, these operations ensure that the correct values are written in the right registers and that the program counter is set to the correct value. The operation ends in state *request instr* and a new cycle begins.

```
 1: property property fetch_16_read_5 is is
 2: for timepoints:
 3:     t_end = t+k;
 4: freeze:
 5:     - - Register values in REG_FILE
 6:     pcReg_at_t = pcReg@t;
 7: assume:
 8:                 - - starting states
 9:     at t+0:     receive_instr;
10:                 - - trigger sequence
11:     at t+0:     (getEncType(loadedDataInstr)=ENC_R);
12:     at t+0:     fromMemoryPort_sync;
13: prove:
14:                 - - output sequence
15: t_end:          pcReg = (4 + pcReg_at_t);
16: during[t+1,t_end]:  fromMemoryPort_notify = false;
17: t_end:          toMemoryPort_sig_addrIn = (4 + pcReg_at_t);
18: t_end:          toMemoryPort_sig_dataIn = 0;
19: during[t+1,t_end-1]:toMemoryPort_notify = false;
20: t_end:          toMemoryPort_notify = true;
21: t_end:          toRegsPort_sig_dst = getRdAddr(INSTR);
22: t_end:          toRegsPort_sig_dstData = getALUresult(...);
23: during[t+1,t_end-1]:toRegsPort_notify = false;
24: t_end:          toRegsPort_notify = true;
25: t_end:          - - Unimportant datapath register ...
26:                         - - ending state
27:     t_end:              request_instr
28: end property;
```

Fig. 19. R-Type instructions

Fig. 19 shows the property generated for R-type instructions. It implements all register-to-register instructions like *add*, *or* and *shift*. The property starts in state *receive instr* (line 9) and it is assumed that a new instruction is available (line 12) from the memory. The encode type of the instruction is evaluated by a combinational function `getEncType()`. A combinational function is not allowed to change any state variables of the module. It returns a value as a function of the input parameters. When the operation is triggered, it is guaranteed that the program counter is set to the correct value (line 15). Lines 17 to 19 set up the request for the next instruction. During the operation all *notify* flags are set to *false* and the memory is notified that there is a new request by raising the *notify* flag at *t_end*. The result of the computation is returned by the combinational function `getALUresult()` and stored to the register file. Lines 21 to 24 sets up the communication with the register file. In line 16, the property ensures that the hardware does not accidentally initiate a read from the memory.

The refinements for the pipelined processor are somewhat more complex. The property describes a single instruction, but, due to the pipelining, the processor executes other instructions in parallel. This results in a more complex refinement of properties and design. Take, for example, the refinement of the source registers. In the sequential implementation it is sufficient to refine the macros for the registers by referring to the datapath registers storing the actual values. In the pipelined implementation data hazards can occur, necessitating implementation of forwarding. The property macros are refined by referring to the values of the forwarding unit or to the datapath registers, depending on whether a hazard was detected or not.

Lastly, the timing in line 3 is refined by providing a value for k which is different for the different implementations:

- *Simple sequential implementation:* $k = 1$. Each operation is executed in one clock cycle.
- *Complex sequential implementation:* $k = 8$. The timing here depends on the communication between the modules. Each operation has its own distinct timing. The designer has to understand the underlying communication sequences and reflect this in the timing.
- *Pipelined implementation:* This is the most complex case, because the timing of the operation depends on the pipeline state, e.g., a stall due to a data hazard increases the time until an operation finishes. In practice, there are two ways to specify this. One way is to keep the value for k static and let the macros describing the pipelining accommodate for the different pipeline states. The other way is to keep the macros static and reflect the pipeline state through different values

for k. In either case, the pipeline state is reflected in the property refinement. The implementation provided online uses the first approach.

TABLE I
DESIGN SIZE AND LOC

Lines of code	ISS	Simple Seq.	Complex Seq.	Pipelined
Properties generated	1165	-	-	-
Properties - lines added	-	0	1	411
Properties - lines changed	-	56	68	56
Implementation	1000	1110	1626	2264
Synthesis				
Input/Output	-	36/71	36/71	36/71
Flip-Flops	-	1340	1881	2698

Table I provides the results for the size of the designs and the property suite. A set of 21 properties was generated from the SystemC-PPA of the ISS in less than 25 seconds. The verification effort in PDD results from the time spent on the refinement of the properties during the implementation process.

Table II provides manual work efforts, CPU times for formal property checking and simulation times for the system-level model. CPU times were measured on an Intel i7-6700 CPU with 32 GB of main memory. Properties were proven with the commercial property checker OneSpin 360 DV [20] and the RTL designs were simulated with ModelSim SE by Mentor Graphics [22]. The second row of the table denotes the design efforts for creating the SystemC model of the ISS and for creating from it the RTL implementations of the different RISC-V implementations. The third row shows the additional manual efforts needed for refining the generated properties during the RTL design process. The reported work efforts were those of first-time users of the methodology.

As can be expected, manual efforts grow as the designs become more complex w.r.t. inter-module communication, timing and pipelining. For example, all operations of the simple processor have a length of one cycle and the design and property refinement starting from the system-level model is nearly trivial. This keeps the work effort for refining the properties under two hours. For the more complex processor versions we exploited the SystemC-PPA communication mechanisms, as discussed in Sec. V, to decompose the ISS model into several SystemC-PPA sub-models that correspond to the different processor modules. The design efforts given in Table II include the efforts for these decomposition steps at the SystemC-PPA level as well as for the creation of the RTL code.

TABLE II
DESIGN EFFORT AND SIMULATION RESULTS FOR DIFFERENT RISC-V IMPLEMENTATIONS

Design and verification results	**ISS**	**Simple Seq.**	**Complex Seq.**	**Pipelined**
Design effort	1 person week	1 person day	4 person days	3 person months
Property refinement effort	—	2 person hours	1 person day	1 person month
Property checking time total	—	2 min	5 min	4:20 h
Longest individual checking time	—	28 s	65 s	1:30 h
Max. memory usage (MB)	—	4003	5220	4628
Simulation time				
Prime numbers (s)	5	16	56	95
Fibonacci (s)	1	4	10	15
Bubble sort (s)	8	35	130	259

The property refinement of the pipelined processor required about 1 person month, due to the complex pipelining. It is important to note that a completed property refinement process implies that all properties hold on the design. Thus, further RTL simulation for verification is not required so that all efforts for traditional simulation and creation of test benches can be avoided.

As shown in Table II, proving the actual properties on the design is very fast, especially for the smaller design. Most properties are proven in less than five minutes. The longest proof time was for the R-type instruction property of the pipelined processor. The complexity lies within the datapath operations, which are a worst-case scenario for SAT engines, due to the large state space. A common practice is to blackbox datapath-heavy components (e.g., the ALU) to reduce proof times drastically.

We simulate the designs with three computation-heavy C++ programs, compiled with the RISCV-V R32 toolchain:

- "Prime numbers", computes ten prime numbers starting from n=10000.
- "Fibonacci", computes numbers of the Fibonacci sequence.
- "BubbleSort", sorts an array with 500 integer numbers. Initially, the numbers are sorted in descending order and the algorithm sorts them in ascending order, resulting in the worst-case execution time for BubbleSort.

The results, as given in Table II, demonstrate a simulation speedup by simulating the ISS between 4 in case of the simple design and ∼32 for the more complex design, compared to the RTL implementations. Simulation of a SystemC model can, obviously, be expected to generally outperform RTL simulation. In case of our PDD methodology the SystemC model, at the abstraction level of an instruction set simulator, has the additional advantage of a sound relationship with the RTL, meaning that the RTL implementation and the ISS execute software in functionally identical ways. This is, to our knowledge, the first time that an ISS can be used as a golden model for design and even for firmware sign-off.

B. SONET/SDH Framer

The second case study is based on an industrial implementation of a SONET/SDH Framer circuit from Alcatel-Lucent. The purpose of the Framer circuit is to identify protocol frames in a serial data stream. A simplified version of the design for illustration purposes is publicly available in [2]. Initially, only an industrial RTL implementation of the Framer was available. We first conducted a complete verification of the design and then created a SystemC-PPA model consisting of two modules, a main module called *Framer* and a supplementary module called *Monitor*. The purpose of the *Monitor* is to collect performance data about the frame detection process realized by the *Framer* module. The work effort for the "bottom-up abstraction" of the RTL design by complete functional verification, as described in [1], was about six person months. Starting with this system described in SystemC-PPA we created an RTL redesign of the circuit by following the PDD methodology. The PDD design process took less than two person months of effort.

TABLE III
SONET/SDH FRAMER — ORIGINAL DESIGN AND REDESIGN

	RTL Design			PPA		
Module	inp./out.	FFs	LoC	inp./out.	var.	states/ops.
Framer (or.)	549/280	4.2k-47k	27k	7/6	4	4/13
Monitor(or.)	20/6	30	850	3/1	2	2/9
Framer (re.)	549/280	3.9k-42k	12k	7/6	4	4/13
Monitor (re.)	20/6	425	92	3/1	2	2/9

Table III shows some statistics for the original design (rows 1 and 2) and the redesign created from PDD (rows 3 and 4). Both designs are configurable through VHDL generics. The actual numbers depend heavily on the configuration parameters. For example, the number of flipflops ranges between ∼4,000 and ∼40,000, depending on the chosen configuration. A large portion of the flipflops (FFs) actually belong to temporary storage needed for buffering the input stream.

Both, the original design and the PDD redesign are sound refinements of the same PPA model. The RTL redesign by PDD has less state variables than the original.

More importantly, though, the code is, subjectively, much "cleaner" than the original implementation which had already seen multiple updates, patches and maintenance changes from several designers. In this sense, PDD can also be seen as a way to deal with legacy code problems, allowing for code cleanup without compromising functional correctness.

More interestingly, however, the new design had a significantly lower power consumption. This has several reasons. First, the "cleaned-up" design has less logic and flipflops than the original one, consuming less power, while still meeting timing and area requirements of the original design. Second, more aggressive optimizations could be explored because their impact on functional correctness could be immediately assessed by running verification of the complete property suite that was created during PDD. An additional improvement of power efficiency would be possible by applying the PPA-based clock gating techniques of [23]. These automatic techniques utilize an existing complete set of properties for creating clock gating circuitry to minimize dynamic power. However, this technique was not employed to gain the above results.

In our experiments, the property suite was generated and refined such that it allowed to describe all design configurations that are possible through the set of VHDL generics provided. The property suite comprises 22 operation properties. Proving them on all possible configurations of the original design takes 23 min and 2 min for the redesign. Proving the most complex property took less than 9 min with a maximum memory consumption of 1589 MB. Memory consumption ranges from 496 MB to 1589 MB for both designs.

TABLE IV
SONET/SDH FRAMER — SIMULATION OF 10^7 FRAMES

Design	RTL sim.	SystemC-PPA sim.	property proofs
RTL Industrial	540 s	2 s	23 min
RTL Redesign	600 s	2 s	10 min

Table IV shows results for the simulation of both the RTL and the SystemC-PPA models. In both simulations, the test bench generated 10^7 SONET frames as input data to the design. A main contribution of PDD is that it enables to move verification

from RTL to the system level. The case study demonstrates that a high degree of abstraction is obtained by the PPA models. In this particular example we can exploit that the correct buffering of the input stream is verified at the RT level and does not need to be represented at the system level. Due to this the simulation times are reduced by a factor of $\sim$270 to $\sim$300, while the soundness of the system model guarantees preservation of the design's I/O sequences.

IX. Conclusion

In this paper we present Property-Driven Development, a novel method to refine transaction-level system models into RTL implementations, along with the supporting open-source tool *DeSCAM*. The underlying theoretical basis of path predicate abstraction ensures a sound formal relationship between the system-level model and the RTL, guaranteeing that verification results obtained at the ESL hold unequivocally also on the RTL. This paves the way to using system-level models as golden references for design. Implementing these models top-down with PDD results in designs that are correct by construction. Additional simulation for design sign-off is not required. PDD is based on the provided tool *DeSCAM*, state-of-the-art property checking and a well-defined methodology for design refinement.

In the first case study, we implemented three different RISC-V CPU designs that are all sound refinements of the same abstract SystemC-PPA model. The refinement effort grows with the complexity of the designs. However, it stays reasonable even for a sophisticated processor design with pipelining and forwarding. The system-level model simulates up to 32 times faster than the RTL implementations. The second case study on an industrial design proved that PDD is practically applicable also to larger-scale designs. It also showed a new way to deal with legacy design problems and demonstrated how a complete formal property suite is usable as a formal specification for a redesign with clean code structure and optimized power consumption.

In conclusion, the presented methodology contributes to shifting global design and verification tasks from the RTL to the system level. By closing the semantic gap between system level and RTL implementation we envision that such a step can make a significant contribution to increase design productivity in future design flows.

References

[1] 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 (TCAD)*, vol. 33, no. 2, pp. 291–304, Feb. 2014.

[2] "DeSCAM," https://github.com/ludwig247/DeSCAM. [Online]. Available: https://github.com/ludwig247/DeSCAM

[3] J. Urdahl, S. Udupi, T. Ludwig, D. Stoffel, and W. Kunz, "Properties first? A new design methodology for hardware, and its perspectives in safety analysis (invited paper)," in *The IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2016.

[4] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 11, pp. 2068–2082, November 2008.

[5] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of TriCore2 and other processors," in *Design & Verification Conference & Exhibition (DVCon)*, 2007.

[6] J. C. Schaaf, Jr. and F. L. Thompson, "System concept development with virtual prototyping," in *Proceedings of the 29th Conference on Winter Simulation*, ser. WSC '97, Washington, DC, USA, 1997, pp. 941–947.

[7] D. Tabakov and M. Y. Vardi, "Monitoring temporal SystemC properties," in *IEEE/ACM Intl. Conf. on Formal Methods and Models for Codesign (MEMOCODE),*, July 2010, pp. 123 –132.

[8] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: a software model checking approach," in *Formal Methods in Computer-Aided Design (FMCAD)*, Austin, TX, 2010, pp. 51–60.

[9] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "KRATOS: a software model checker for SystemC," in *Intl. Conf. on Computer-Aided Verification (CAV)*, ser. CAV'11. Springer, 2011, pp. 310–316.

[10] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *Proc. Intl. Design Automation Conference (DAC)*, 2013, pp. 116:1–116:6.

[11] A. Kölbl, J. R. Burch, and C. Pixley, "Memory modeling in ESL-RTL equivalence checking," in *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, 2007, pp. 205–209.

[12] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to RTL equivalence checking," in *Design, Automation Test in Europe Conference (DATE)*, april 2009, pp. 196 –201.

[13] L. Foundation, "LLVM," http://llvm.org. [Online]. Available: http://llvm.org

[14] A. Kaushik and H. D. Patel, "SystemC-Clang: An open-source framework for analyzing mixed-abstraction SystemC models," in *Forum on Specification & Design Languages (FDL)*, Sept 2013, pp. 1–8.

[15] K. Marquet and M. Moy, "PinaVM: A SystemC front-end based on an executable intermediate representation," in *Proc. Intl. Conf. on Embedded Software (EMSOFT)*, 2010, pp. 79–88.

[16] F. Forschungszentrum für Informatik, "KaSCPar," ftp.fzi.de/downloads/sim/archives/kascpar-documentation.pdf. [Online]. Available: ftp.fzi.de/downloads/sim/archives/kascpar-documentation.pdf

[17] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken, and R. Wille, "Completeness-driven development," in *Graph Transformationsw*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 38–50.

[18] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proc. International Design Automation Conference (DAC)*, June 1999, pp. 317–320.

[19] G. J. Holzmann, "The SPIN model checker," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, 1997.

[20] Onespin Solutions GmbH, "OneSpin 360 DV-Verify," https://www.onespin.com/products/360-dv-verify/.

[21] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual volume II: Privileged architecture version 1.9," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-161*, 2016.

[22] Mentor Graphics, https://www.mentor.com/.

[23] S. Udupi, J. Urdahl, D. Stoffel, and W. Kunz, "Dynamic power optimization based on formal property checking of operations," in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, 2017.