Formal Techniques for Effective Co-verification of Hardware/Software Co-designs

Rajdeep Mukherjee
University of Oxford
rajdeep.mukherjee@cs.ox.ac.uk

Raphael Polig
IBM Research, Zurich
pol@zurich.ibm.com

Mitra Purandare
IBM Research, Zurich
mpu@zurich.ibm.com

Daniel Kroening
University of Oxford
kroening@cs.ox.ac.uk

ABSTRACT

Verification is indispensable for building reliable of hardware/software co-designs. However, the scope of formal methods in this domain is limited. This is attributed to the lack of unified property specification languages, the semantic gap between hardware and software components, and the lack of verifiers that support both C and Verilog/VHDL. To address these limitations, we present an approach that uses a bounded co-verification tool, HW-CBMC, for formally validating hardware/software co-designs written in Verilog and C. Properties are expressed in C enriched with special-purpose primitives that capture temporal correlation between hardware and software events. We present an industrial case-study, proving bounded safety properties as well as discovering critical co-design bugs on a large and complex text analytics FPGA accelerator from IBM$^{\circledR}$.

Keywords

HW/SW Co-verification, Firmware, Verilog, Text Accelerator Co-design, SAT/SMT Solver, Symbolic Execution

1. INTRODUCTION

The ever-increasing complexity of SoCs and tighter integration of hardware (HW) and software (SW) components makes a strong case for formal co-verification techniques. Correct functionality of the co-design requires verification of not only the individual HW and SW components but also their complex interplay. The challenges in co-verification are manifold: 1) lack of means to specify properties over the co-design, 2) the semantic gap between the synchronous clock-driven HW and the asynchronous event-driven SW, and 3) lack of co-verification tools that support co-designs, which are typically written in a combination of C and Verilog/VHDL.

A key to building an effective co-verification tool is to translate the HW and SW models to a common representation with formal semantics, which we call *co-verification model*, enabling specification of system-level properties across HW/SW boundaries. The co-verification model must be able to exploit the expressivity of the underlying reasoning engines: a joint bit-level model is easy to construct but is an ill-fit for word-level Satisfiability Modulo Theory (SMT) solvers. In this paper, we perform symbolic execution of the HW and SW in tandem to generate the co-verification model. We can synthesize either a bit-level netlist (represented as And-Inverter Graph) or word-level netlist (represented in a format that resembles the SMT-LIB standard) from the HW given in RTL Verilog. Similarly, the SW, given in C, is automatically translated into a Static Single Assignment (SSA) form. A monolithic verification condition is generated from these two models, which is then solved using a satisfiability (SAT) or SMT solver. A co-design property is specified in C with the support of a special primitive, called *next_timeframe()*, which can express temporal correlation between HW and SW events. Our co-design properties are expressive, amenable to verification reuse and can be readily used in assertion-based verification. Additionally, our tool also supports a subset of the IEEE Property Specification Language (PSL), which formally captures design intent for HW given in Verilog RTL.

Our technique is embodied in our HW/SW co-verification tool, HW-CBMC [7]. HW-CBMC performs bounded model checking of reachability properties of HW/SW co-designs. The HW/SW co-designs can be specified in Verilog RTL and C, respectively.

We report an industrial application of HW-CBMC on the Text Analytics FPGA Accelerator (TAA) co-design [14, 13]. Text analytics is key in big data analytics. Heterogeneous computing platforms involving CPU and FPGA are popular in big data analytics. However, verification of such platforms has received little attention. The TAA, as illustrated in Figure 1, is an FPGA-based streaming text analyzer, shown to increase the throughput of the information extraction system called SystemT [13]. The TAA is a co-design in which the SW sends documents or streams of data packets to the FPGA for analysis and reacts to the response of the HW. The FPGA processes text analytics queries, which mostly consist of regular expression matching. The HW in the TAA is a Verilog RTL implementation and the SW is written in C. Formally verifying the TAA co-design poses a challenge mainly due to the semantic gap between the FPGA accelerator and the SW, the lack of co-specification, the interleaving state-space of the HW and SW threads, and the sheer complexity of the whole system.

Contribution: Our contributions are three-fold:

1. We present an automatic, bounded HW/SW co-verification tool HW-CBMC, which formally validates co-designs written in Verilog RTL and C, respectively.

2. A unified property specification framework for HW/SW co-verification is presented. The framework allows specification of properties over hardware and software events and their

Figure 1: Overview of the Text Analytics Accelerator design

Figure 2: HW-CBMC Tool Flow

temporal correlations. Our tool also supports a subset of IEEE Property Specification Language (PSL).

3. The tool has been evaluated on the Text Analytics FPGA Accelerator [14, 13] co-design from IBM®. The experimental results demonstrate that HW-CBMC automatically uncovered several critical co-design bugs and proved several complex properties of the TAA in less than 15 minutes.

The rest of the paper is organized as follows. Section 2 explains the verification framework. The experimental evaluation appears in Section 3. Section 4 explains related work. Section 5 concludes.

2. VERIFICATION FRAMEWORK

2.1 Front-end

HW-CBMC supports HW design in IEEE 1364-2005 System Verilog standard and SW implementations in C89, C99 and C11. In a typical SoC environment, SW communicates with the HW through its input/output ports. Given a HW design in Verilog RTL, HW-CBMC automatically generates an interface module in C consisting of all HW input/output/inout port signals. The interface module enables the SW to communicate with the HW ports through special primitives called *set_inputs*() and *next_timeframe*(). When the SW executes *set_inputs*(), it directs HW-CBMC to set the HW ports to the values assigned by the SW. A call to *next_timeframe*() advances the HW clock. Section 3 demonstrate the use of these handshake primitives on the TAA co-design.

2.2 Generating Unified Co-verification Model

Figure 2 illustrates the architecture of HW-CBMC. A unified co-verification model is generated by translating the HW and SW semantics to a common formal semantics. The top flow in Figure 2 synthesizes the input HW design given in Verilog RTL to either a bit-level netlist or a word-level netlist. The bit-level netlist is represented as an AIG, whereas the word-level netlist is represented in a format that resembles the SMT-LIB standard. The bottom flow in Figure 2 translates the SW given in C into static single assignment (SSA) form. A unified monolithic verification condition is then generated from the SSA and HW netlist. The verification condition is represented in either DIMACS or SMT-LIB format, which is then checked with a SAT or SMT solver, respectively.

2.3 Analysing HW/SW Interaction Patterns

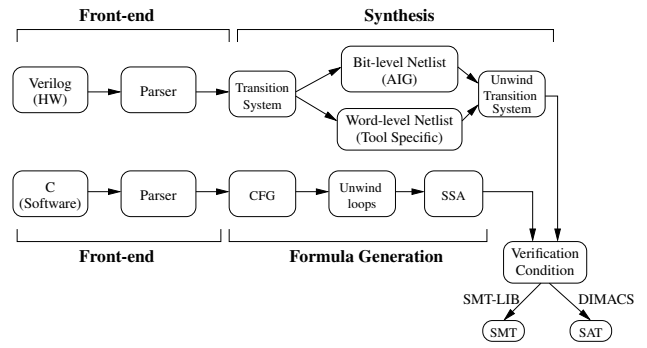A system comprises a set of concurrent SW and HW threads, which interleave asynchronously. Concurrency is a key challenge in co-verification. The number of interleavings between the SW and HW threads can be extremely large, which may become a bottleneck in the verification process. We formalize the communication between HW and SW using *transactions*. Transactions allow us to decompose the HW/SW interleaving state-space into easily verifiable service-level [1] interactions, thereby reducing the number of interleavings.

Software Transaction.

A Software Transaction ($\mathcal{ST}$) is an untimed state machine with a unique start state and an unique end state. Transitions in $\mathcal{ST}$ may be guarded with a quantifier free predicates over SW variables or HW port signals (for example, acknowledgement received from HW). If the guard evaluates to TRUE, it may perform some action, which may be read/write into shared memory or cacheline or HW ports. If the guard is not specified, it is assumed to be TRUE. $\mathcal{ST}$ is often manifested as control-flow paths in the SW. An execution of $\mathcal{ST}$, also called a SW transaction instance, is a simple path from start state to end state. Repeated execution of $\mathcal{ST}$ can happen only in a sequential manner.

Hardware Transaction.

Consider a HW RTL design with a *inputs* $\{I\}$, *outputs* $\{O\}$, *wires* $\{Wire\}$ and a set of state elements such as *Registers* $\{Reg\}$. RTL designs also support memory elements which is a sequence of registers. Note that $\{I\} \subseteq \{Wire\}$, and $\{O\} \subseteq \{Reg\} \cup \{Wire\}$. All state elements are fixed-width unsigned integers. An *update* of RTL design changes the values of $\{Reg\}$ and $\{Wire\}$.

DEFINITION 1. *A transaction step φ is defined by a pair $\varphi \equiv \langle \Gamma_\varphi, \Delta_\varphi \rangle$. Here, $\Gamma_\varphi \equiv \{\alpha_1, \alpha_2, \ldots \alpha_m\}$ is a set of updates to state elements such as $\{Reg\}$ where α_i is an update at the i-th transaction step. Whereas, $\Delta_\varphi \equiv \{\beta_1, \beta_2, \ldots \beta_n\}$ is a set of combinational updates to $\{Wire\}$, where β_j is an update at the j-th transaction step.*

DEFINITION 2. *A Hardware Transaction $\mathcal{HT} = (V, \Sigma, \rightarrow, init, comp)$ is a labelled transition system, where V is a set of transaction steps φ with a unique source vertex, $init \in V$, and a unique sink vertex, $comp \in V$, $\Sigma \subseteq 2^{\mathcal{G}}$ is a set of labels where $\mathcal{G}$ is a set of quantifier free Boolean predicates over α_i and β_j and $\rightarrow \subseteq V \times \Sigma \times V$ is a set of labelled transitions.*

Each execution of $\mathcal{HT}$ is a non-empty sequence of transaction steps where each transaction step happens in a single clock. The first step is called *initiation event* (*init*). Assuming the duration of $\mathcal{HT}$ is finite, there must exist some distinguishable last completion event (*comp*). After a transaction step φ is executed, the set of state
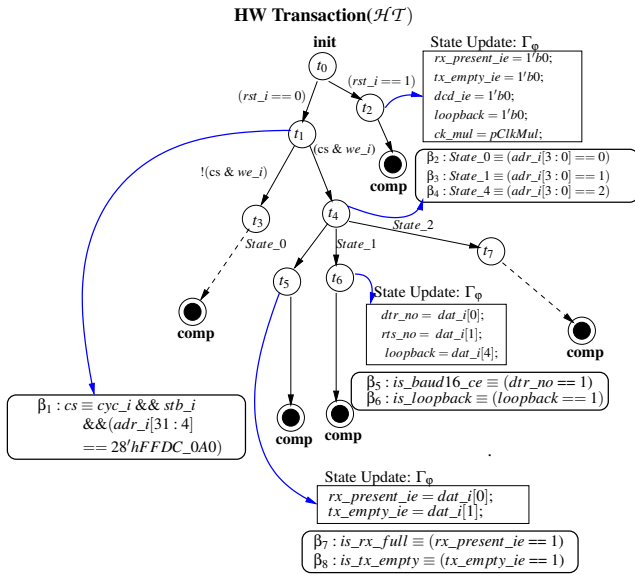
Figure 3: Some HW Transactions for RTL block in TAA

update in Γ_φ and combinational updates in Δ_φ is carried out and the value of Boolean predicates in $\mathcal{G}$ is evaluated to determine the next step φ_{next}, that is, a predicate over $\beta_i \in \Delta_\varphi$ and $\alpha_j \in \Gamma_\varphi$ enables a transaction step. This process continues until $\mathcal{HT}$ reaches the *comp* state. Closest to our notion of $\mathcal{HT}$ is the micro-architecture-level transaction by Mahajan et al. [10].

Figure 3 presents a fragment of $\mathcal{HT}$ corresponding to a RTL block of TAA design. Let us consider a possible execution of $\mathcal{HT}$, $(init \rightarrow t_1 \rightarrow t_4 \rightarrow t_6 \rightarrow comp)$. At *init*, the SW sets the RTL input signal, $(rst\_i = 0)$, based on which the execution in $\mathcal{HT}$ follows the edge to t_1. At t_1, a combinational update β_1 is performed which updates cs based on HW input signals, cyc_i, stb_i, adr_i and we_i. Assuming that $(cs \ \&\& \ we\_i)$ holds true, the execution proceeds to step t_4 which performs some combinational updates $(\beta_2, \beta_3, \beta_4)$ based on the values in adr_i. The next step is determined by the predicates over β_2, β_3 and β_4. Assuming that β_2 holds true, the execution proceeds to t_6. During the execution of t_6, three state update $(dtr\_no, rts\_no, loopback)$ and two combinational updates (β_5, β_6) are performed. After t_6, $\mathcal{HT}$ leads to *comp* and the transaction finishes its execution. The labels in the edges corresponds to the Boolean predicates $\mathcal{G}$ computed over elements in Γ_φ and Δ_φ. The blue arrows point to the state updates or combinational updates at a particular transaction step φ.

We must emphasize that a single transaction instance in $\mathcal{ST}$ may trigger sequence of transaction steps in a $\mathcal{HT}$ that spans several clock cycles as is evident from this example.

Producer-consumer Interaction.

An interaction is said to exhibit *producer-consumer* interaction pattern if (1) the set of variables V written by the execution of $\mathcal{ST}$ and and the set of variables V' written by the execution of $\mathcal{HT}$ are disjoint $(V \cap V' = \emptyset)$, (2) and the executions of $\mathcal{ST}$ and $\mathcal{HT}$ consumes values in V' and V, respectively.

The TAA co-design exhibits a producer-consumer relationship in which the SW and HW transactions never write to the same virtual memory space at the same time. When $\mathcal{ST}$ writes to a shared address space, $\mathcal{HT}$ reads from that shared space. Similarly, when $\mathcal{HT}$ updates a shared memory or a cacheline, $\mathcal{ST}$ consumes the data from that shared space. We also observed similar producer-

consumer interaction patterns in other co-designs, such as coherent FFT coprocessor [4].

2.4 Property Specification for Co-verification

The lack of unified property specification language has been a major bottleneck for the application of formal techniques for effective co-verification. To address this challenge, HW-CBMC supports a unified property specification in C with special primitives that allow verification engineers to write temporal assertions over both HW and SW events. Closest to our unified property specification for co-verification is the language *xPSL* [17] proposed by Xie et al., which is the extension of IEEE PSL. Additionally, HW-CBMC also supports a subset of IEEE PSL that includes immediate and concurrent assertions except sequences. In this paper, we classify co-design properties into two classes – *transaction-level* property and *component-level* property.

Transaction-level Property.

A *Transaction-level* property (ψ_t) formally captures the design intent of the interaction between a $\mathcal{ST}$ and $\mathcal{HT}$, that is, it is defined at the transaction boundaries. Thus, a transaction-level property follows implication structure – the antecedent of the implication is an event in $\mathcal{ST}$ or $\mathcal{HT}$ and the consequent is an event in $\mathcal{HT}$ or $\mathcal{ST}$ respectively. An event in $\mathcal{ST}$ may be a function call or an update to program variables. Similarly, an event in $\mathcal{HT}$ corresponds to an update of combinational or sequential elements. Figure 4 presents some examples of transaction-level properties for the TAA co-design. Note that the HW events are in marked bold to distinguish it from SW events. The properties in figure 4 capture different segments of the waveform in Figure 7. For example, property $P1.1$ and $P2.1$ corresponds to the segment $W1$ and $W2$ respectively.

A SW event may trigger a sequence of transaction steps in a HW. This is handled by writing monitors (in C) which keeps track of the number of clock cycles elapsed between the two events. For example, consider the property $P3.2$ shown in Figure 4. When *afu_ta_clear_halt*() has done a MMIO write, the HW signals *halt* and *hw_rslt_ea* is set to LOW and $A0$ respectively in the next cycle. HW-CBMC provides a *next_timeframe*() primitive to keep track of the HW clock. Thus, whenever the SW event, *afu_ta_clear_halt*(), is successfully complete, the monitor calls a *next_timeframe*() before asserting the resultant HW signals (*halt*, *hw_rslt_ea*).

Monitors (in C)	

```// afu_ta_submit() returns 1```
```// on successful job submit```
```int monitor_P1.1() {```
` assert(!afu_ta_submit()`
` || (mm_valid == 1) );`
```}```
	```// afu_ta_read() is a HW```
	```// requested read operation```
	```int monitor_P2.3() {```
	` assert(!afu_ta_read() ||`
	` ((get0_data_r&0x1)==1));`
	```}```

```// afu_ta_wait() returns 1```
```// on successful exit```
```// from waiting```
```int monitor_P3.1() {```
` assert( !(put_data_v==1`
` && hw_rslt_ea==STS)`
` || afu_ta_wait());`
```}```
	```int monitor_P3.2() {```
	` if(afu_ta_clar_halt()){`
	`  next_timeframe();`
	`  assert( halt==0`
	`  && hw_rslt_ea==A0 );}}`
	```int monitor_P3.3() {```
	` if(hw_req_read_bufs()){`
	`  next_timeframe();`
	`  next_timeframe();`
	`  assert( data_ack == 1 );`
	` }}```

Figure 4: Transaction-level property as monitor (in C)

Component-level Property.

A *component-level* property (ψ_c) is defined purely at the level of individual components of a co-design system. Recall that a

component may be a HW model, a SW model or an interface model which contains only the interface logic. A component-level property captures the design intent of a component in terms of its input/output behaviors. This type of property is useful in a co-design environment for verifying individual components. Note that unlike the structure of ψ_t which requires the antecedent or consequent to be a HW or SW event, ψ_c only contains signals (ports) or variables of a particular component. Figure 5 presents some examples of component-level properties for the accelerator core unit. The properties are expressed as monitors (in C) as shown on the right side of Figure 5. The equivalent System Verilog Assertion (SVA) are shown on the left side of Figure 5.

System Verilog Assertions	Monitor (in C)
P1.2: assert property (@(**posedge** clk) mm_ack == 1|−> (mm_valid == 1 && ##2 job_submit == 1));	int monitor_P1.2() { assert(!mm_ack || mm_valid); next_timeframe(); next_timeframe(); assert(job_submit); }
P2.1: assert property (@(**posedge** clk) (ctrl_state == 1 && get_v && get_r) |−> ##1 (ctrl_state == 2)); P2.2: assert property (@(**posedge** clk) (get_data_v) |=> ##2 (ctrl_state == 3));	int monitor_P2.1() { if(ctrl_state ==1 && get_v && get_r) { next_timeframe(); assert(ctrl_state == 2); } } int monitor_P2.2() { if(get_data_v) { next_timeframe(); next_timeframe(); assert(ctrl_state == 3); } }

Figure 5: Component-level property (in SVA) as monitor (in C)

2.5 Debugging using Counterexample Trace

A bug in a co-design environment may occur due to defective HW or erroneous SW implementation or both. To ease the process of localizing bugs in co-verification, HW-CBMC generates a counterexample trace up to a user-specified bound in case of property failure. The trace contains concrete assignments to SW variables, interface signals and HW RTL signals for each timeframe that leads to the bug. The counterexample trace is generated in the value change dump (.vcd) format, which can be viewed graphically using a waveform viewer.

3. EXPERIMENTAL RESULTS

We report experimental results for HW/SW co-verification of the proprietary Text Analytics FPGA Accelerator co-design using our tool HW-CBMC. All our experiments were performed on an Intel® Xeon 3.40 GHz machine with 32 GB RAM. All times and memory consumption are reported in seconds and Megabytes respectively.

State-of-the-art Co-verification tools: Building automated co-verification tools has received little attention in the past. We are not aware of any automated formal co-verification tool that can readily accept co-designs written in C and Verilog RTL. In this paper, we present HW-CBMC to address this gap.

3.1 Case Study: Text Analytics FPGA Accelerator Co-design

Figure 1 illustrates the overall architecture of TAA co-design. It consists of a core component (dotted outline), which is responsible for the communication with the SW and is the key HW component to be verified. The core uses an IP block for the POWER Service Layer (PSL) to implement the CAPI protocol. The PSL is black boxed and assumed to operate correctly. On the other side, the core uses the Query Core (QC) to process the input data and generate results.
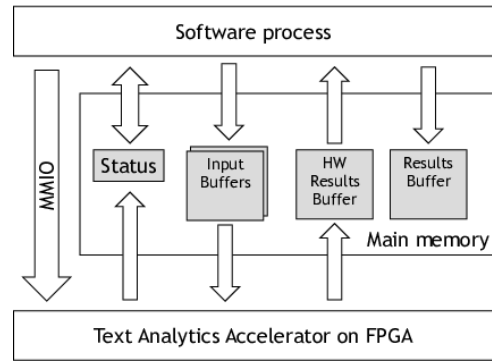

Figure 6: Communication between TAA FPGA and SW through the virtual memory space

The QC is a custom IP block that is different for every defined query. The correctness of the results from the QC is verified in a separate step. Our co-verification effort focus on the communication between the core and the SW. For this purpose, the QC is replaced by a greybox, which implements the communication protocol and generates a predefined number of random results that need to be written back to memory.

The TAA consists of 15879 registers, 8886 ALUTs (Adaptive LUT) and two memory blocks on the FPGA, each of 16 Kb each. The design statistics are obtained by synthesis for an Altera Stratix V FPGA.

The TAA leverages the Coherent Accelerator-Processor Interface (CAPI) [16], enabling the accelerator to operate in the virtual memory space of a process just like a SW thread on the CPU. Because the accelerator operates on memory regions shared with the SW, similar mechanisms have to be implemented to avoid both SW and HW accessing the same data at the same time. Thus, verification of the access control mechanism is needed to prevent data corruption, data loss, or faulty executions.

Figure 6 illustrates the various data structures accessed by the SW process and the accelerator. Furthermore, the SW can push small amounts of data (up to 64 bits) via memory-mapped I/O operations (MMIO) to the accelerator. All the data structures are grouped into a structure called *work element descriptor* (WED). The SW prepares the WEDs. The WED holds a status field, which is used to notify the SW if the accelerator has completed processing or requires help. The input buffers are only read by the accelerator and results are written into a HW results buffer. This buffer has a fixed size, which is also stored in the WED. As the number of results cannot be determined beforehand, the accelerator fills this buffer until it is full. The SW can then fetch the results from the HW buffer and put them into a final result buffer, which it can re-size if required. The text analytics FPGA accelerator interacts with a SW that sends documents or streams of data packets to the FPGA for analysis. The left-hand side of Fig. 8 gives a simplified code-snippet of the SW that implements the orchestration strategy for interacting with the FPGA. The interface functions, given on the right-hand side of Fig. 8, are the lowest level of implementation that accesses the HW ports. The access to the HW ports is done through a global structure, *afu_ta_core*, which contains all the input/output ports of the top-level RTL module of text analytics accelerator. The SW transactions are highlighted in *red*, followed by calls to monitors (marked in *blue*), that checks the validity of a transaction.

Table 1 presents the detailed results for HW/SW co-verification of the text analytics co-design. Figure 4 and Figure 5 gives the
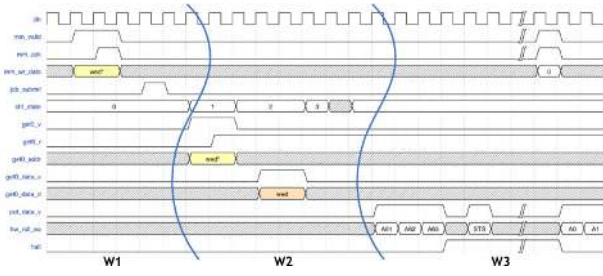
Figure 7: Waveform of the accelerator core when W1) receiving a WED pointer, W2) requesting and receiving the actual WED and W3) writing results until the results buffer is full and is cleared by SW before processing continues.

properties under verification. Column [1-12] in Table 1 report the property id, type of property, the bound up to which the HW transition system is unwound, the unwind depth for the SW, number of SAT clauses, number of variables, the verification result (*safe* or *unsafe*), the source of the bug (HW or SW or Interface logic) if *unsafe*, and the total verification run time and memory consumption for bit-level and word-level backend solvers, respectively. Note that the value of *bound* must be greater or equal to the number of *next_timeframe*() calls. The *unwind* depth is determined by the highest bound among all bounded loops in the SW.

Bit-level versus Word-level Verification: HW-CBMC offers two different verification backends – 1) *bit-level* and 2) *word-level*. We configured HW-CBMC with MiniSAT-2.2.0 [3] and Z3 [2] for bit-level and word-level reasoning, respectively, for our experiments. The theories used for SMT solvers are theory of bit-vector, arrays and uninterpreted functions. Though the runtimes for both the backends are very close to each other, the bit-level backend performed marginally better than the word-level backend.

Proving bounded safety: We proved several properties of the TAA co-design. Table 1 gives runtimes for proving two transaction-level properties ($P1.1$, $P3.1$) and two component-level properties ($P1.2$, $P2.2$). The property $\psi_t = P1.1$ in Figure 4 specifies that when the SW executes $afu\_ta\_submit()$, it performs a MMIO write to a specific offset address on the HW, thereby setting the HW signal mm_valid to HIGH. On the other hand, $P1.2$ and $P2.2$ are purely RTL properties that check the correctness of the accelerator core and controller finite state machine (FSM) respectively. Refer to waveform segment W1, W2 and W3 of Figure 7 for a pictorial illustration of {$P1.1$, $P1.2$}, {$P2.2$} and {$P3.1$}, respectively.

Bug hunting: We identified four critical bugs – three transaction-level bugs ($P2.3$, $P3.2$, $P3.3$) and one component-level bug ($P2.1$). Due to the space limit, we only explain one scenario, $P3.3$. This bug is triggered when the FPGA accelerator attempts to read the WED data structure from the main memory. The data channel that is used for this transfer is shared between the FSM control logic and GET0 module which fetches the document data after the WED has been transferred. Thus, the HW components that control the flow control (data acknowledge) on this channel are – *core FSM* and *document GET0 module*. While the FSM signals enable the registers that capture the WED, the flow control has been fully overtaken by GET0 module. In such a scenario, the data-flow would never advance because the data is never acknowledged. This scenario occurs due to a bug in the combinational logic of the shared data channel which always pass the flow control to the GET0 module. If GET0 receives backpressure from the query core it never acknowledge the read for the WED data and the entire processing is

```
Main Function                          | Interface Functions
// WED structure                       | /* ====== IDLE ========*/
struct afu_ta_wed {                    | void idle() {
  uint8_t  *doc;                       |   set_inputs();
  uint32_t docSize;                    |   next_timeframe();
  uint32_t *sts;                       | }
  uint32_t *buf;                       | /* ====== MMIO WRITE ======*/
  uint32_t bufSize;                    | void core_mmio_write
}                                      |     (int offset, uint64_t data)
                                       | {
int main(void)                         |   offset = offset >> 2;
{                                      |   afu_ta_core.mm_valid = 1;
  // Allocate the WED structure        |   afu_ta_core.mm_rnw = 0;
  struct afu_ta_wed *wed;              |   afu_ta_core.mm_dw = 1;
  int afu_ret = 0;                     |   afu_ta_core.mm_addr = offset;
  afu_ta_init(&wed);                   |   afu_ta_core.mm_wr_data = data;
                                       |   set_inputs();
  uint32_t *results, result_block = 0; |   idle();
  results = malloc(wed->bufsSize);     |   afu_ta_core.mm_valid = 0;
  ******************                   |   set_inputs();
  afu_ta_start();                      | }
  ******************                   | void mmio_write64(
  assert(!afu_ta_start() || reset == 0); | struct cxl_afu_h *afu_h,
  ******************                   | int offset, uint64_t data) {
  afu_ta_submit(wed);                  |   core_mmio_write(offset, data);
  ******************                   | }
  monitor_P1.1();                      | /* ====== AFU_TA START ======*/
  // Wait for HW to become active      | int afu_ta_start(){
  for(int i=0;i<=15;i++) {             |   afu_ta_core.reset = 1;
   if(afu_ta_core.states == 0)         |   idle();
    idle();                            |   afu_ta_core.reset = 0;
   else break;                         |   afu_ta_core.mm_valid = 0;
  }                                    |   afu_ta_core.get0_r = 0;
  while(afu_ret!=0) {                  |   idle();
  ******************************       | }
  afu_ret = afu_ta_wait(wed);          | /* ===== AFU_TA SUBMIT =======*/
  afu_ta_copy_results(wed, results);   | int afu_ta_submit(
  ******************************       | struct afu_ta_wed *wed) {
  monitor_P2.3();                      |   mmio_write64(afu_h, 0x80,
                                       |            (uint64_t) wed);
  if(afu_ret>1) {                      | }
    // Increase result buffer          | /* ===== AFU_TA CLEAR =======*/
    result_block++;                    | int afu_ta_clear_halt(uint32_t
    ****************************        | threadId, struct afu_ta_wed *wed)
    afu_ta_increase_buffer(wed, results, | {
    result_block);                     |   wed->sts[0] = 0;
    afu_ta_clear_halt(wed, afu_ret);   |   if(streamid==36)
    ****************************        |     mmio_write64(afu_h, 0x90, 0);
    monitor_P3.2();                    |   if(streamid==38)
  } else {                             |     mmio_write64(afu_h, 0x98, 0);
    break;}                            |   if(streamid==40)
  ***********************              |     mmio_write64(afu_h, 0xa0, 0);
  hw_req_read_bufs(wed);               |   if(streamid==42)
  ***********************              |     mmio_write64(afu_h, 0xa8, 0);
  monitor_P3.3();                      | }
  }                                    |
}                                      |
```

Figure 8: Code snippet of SW model interacting with TAA HW

blocked. The bug is manifested in the FPGA HW with a unrolling bound of 150. The counterexample trace provided by HW-CBMC help to localize the bug and fix the orchestration strategy on the data channel.

3.2 Challenges and Experiences

We describe various challenges and experiences for formal verification of text analytics co-design. It is worth mentioning that the simulation-based verification using a commercial simulator could not cover all possible testcases due to the large number of possible interleavings between the SW and the HW components. However, the application of the formal co-verification tool HW-CBMC automatically detected several critical bugs. Majority of these bugs are either because of the divergence of HW RTL from the behavior expected by the SW or due to a timing mismatch in the low-level SW interface that interacts with the RTL.

One of the most important tasks in our co-verification effort is to write meaningful properties that cover all possible scenarios and interleavings between the SW and the text analytics HW. A total of 23 transaction-level properties are verified for the TAA co-design. Another 15 component-level properties are verified to determine the correctness of the SW or the HW FPGA accelerator in a stand alone manner. The maximum time taken for verifying each property is approximately 15 minutes. Due to the space limit, we report only eight representative properties in this paper.

The subsequent task is to systematically black-box and grey-box the RTL blocks such as the POWER Service Layer that do not

Table 1: HW/SW Co-verification of Text Analytics Co-design

Property	Property Type	Bound (HW)	Unwind (SW)	#Clauses	#Variables	Verification Result	Bug Type	Bit-level Netlist		Word-level Netlist	
								Time (sec)	Memory (MB)	Time (sec)	Memory (MB)
P1.1	ψ_t	20	10	7744051	3441174	SAFE	–	137.5	3125.4	142.6	3128.2
P1.2	ψ_c	40	15	8325191	4190234	SAFE	–	488.3	4287.6	492.2	4261.7
P2.1	ψ_c	60	25	8389278	4202189	UNSAFE	Hardware	583.4	4675.4	588.9	4562.8
P2.2	ψ_c	70	35	8976345	3894572	SAFE	–	543.8	4785.3	578.5	4806.7
P2.3	ψ_t	90	40	9231568	5673214	UNSAFE	Software	619.2	5183.9	623.3	5297.4
P3.1	ψ_t	95	60	9567821	5789213	SAFE	–	629.6	5321.9	667.3	5342.7
P3.2	ψ_t	100	70	9646713	5986743	UNSAFE	Hardware	685.7	5561.5	678.3	5568.9
P3.3	ψ_t	150	80	9784516	6342518	UNSAFE	Hardware	717.4	5921.9	778.2	5921.6

impact the verification of the text analytics engine. For scalable and effective co-verification, the HW/SW interaction behavior is analysed to decompose the complex interleaving into simple and easily verifiable interaction patterns. Recall that the TAA co-design exhibits producer-consumer relationship. This helps us to identify those events (*transactions*) that drive a component to a desired state. Note that such a *pre-defined* state is derived from the *operation manual* of the TAA co-design. The advantage of such decomposition is two fold – *scalability* and *faster bug hunting*. The improved performance is mainly attributed to the fine granularity of properties that captures the design intent for shorter interaction sequence between the SW and the FPGA accelerator. In case of property failure, a counterexample trace is provided by HW-CBMC which contains the SW and HW traces that leads to the bug. The counterexample trace is analyzed to localize the source of the bug.

4. RELATED WORK

Previous work [1, 6] addresses co-verification in the presence of a Transaction Level Model (TLM) of HW. Malik et al. [1] have developed a methodology for firmware validation using a service function-based TLM that models both firmware and its interacting HW component. The work of [8, 17] concerns co-verification for the case of an RTL HW. Unified high-level HW/SW models for co-verification have been pursued in the past [9]. Notably, Monni-aux [11] model HW and SW as C programs, which are formalized as pushdown systems (PDS), Li et al. [8] use Büchi Automata to model HW and a PDS to model SW to generate a unified model, called Büchi Pushdown System (*BPDS*). The work of [15, 5, 12] performs co-verification using abstraction techniques. To assist co-verification at a high level of abstraction, Xie et al. [17] have developed a property-specification language called *xPSL*, which extends IEEE PSL to temporal assertions over HW and SW events.

5. CONCLUSION

We have presented HW-CBMC, a HW/SW co-verification tool that formally validates co-designs written in C and Verilog RTL. We also presented an unified property specification framework for co-verification that can express temporal assertions over SW and HW events. To enable effective co-verification, we decompose the interleavings between HW and SW into transaction-level properties, thereby allowing faster detection and localization of bugs.

We demonstrated bounded HW/SW co-verification of the text analytics FPGA accelerator co-design from IBM® using HW-CBMC. We proved several complex properties of the text analytics co-design. We found several critical co-design bugs using our tool in less than 15 minutes. There are several further directions to explore. In future, we plan to extend the applicability of our verification methodology to other industrial co-designs. An interesting direction is to explore other decomposition techniques to reduce the size of the BMC formula. Another possible direction is to explore unbounded proof techniques in HW/SW co-verification.

6. REFERENCES

[1] S. Ahn and S. Malik. Automated firmware testing using firmware-hardware interaction patterns. In *CODES+ISSS*, pages 1–25, 2014.

[2] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.

[3] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[4] H. Giefers, R. Polig, and C. Hagleitner. Accelerating arithmetic kernels with coherent attached FPGA coprocessors. In *DATE*, pages 1072–1077, 2015.

[5] D. Große, U. Kühne, and R. Drechsler. HW/SW co-verification of embedded systems using bounded model checking. In *GLSVLSI*, pages 43–48, 2006.

[6] A. Horn, M. Tautschnig, C. G. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening. Formal co-validation of low-level hardware/software interfaces. In *FMCAD*, pages 121–128, 2013.

[7] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371, 2003.

[8] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey. An automata-theoretic approach to hardware/software co-verification. In *FASE*, pages 248–262, 2010.

[9] F. Lugou, L. Apvrille, and A. Francillon. Toward a Methodology for Unified Verification of Hardware/Software Co-designs. *Journal of Cryptographic Engineering*, pages 1–12, 2016.

[10] Y. Mahajan, C. Chan, A. Bayazit, S. Malik, and W. Qin. Verification driven formal architecture and microarchitecture modeling. In *MEMOCODE*, pages 123–132, 2007.

[11] D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In *EMSOFT*, pages 30–36, 2007.

[12] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz. Formal hardware/software co-verification by interval property checking with abstraction. In *DAC*, pages 510–515, 2011.

[13] R. Polig, K. Atasu, L. Chiticariu, C. Hagleitner, H. P. Hofstee, F. R. Reiss, H. Zhu, and E. Sitaridi. Giving text analytics a boost. *Micro*, 34(4):6–14, 2014.

[14] R. Polig, K. Atasu, H. Giefers, and L. Chiticariu. Compiling text analytics queries to FPGAs. In *FPL*, pages 1–6, 2014.

[15] S. Swan. SystemC transaction level models and RTL verification. In *DAC*, pages 90–92, 2006.

[16] B. Wile. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. IBM White Paper, Sep 2014.

[17] F. Xie and H. Liu. Unified property specification for hardware/software co-verification. In *COMPSAC*, volume 1, pages 483–490, July 2007.