ABSTRACT

Ensuring functional correctness of hardware and software is a bottleneck in every design process of Embedded Systems. This paper proposes an approach to formally verify low-level software in conjunction with the hardware. The proposed approach is based on Interval Property Checking (IPC) that has proved successful on large industrial hardware designs. In this paper, IPC is extended by a specific abstraction technique that makes it tractable for hardware/software co-verification on realistic industrial designs. In the proposed methodology sets of finite state sequences of the system are abstracted by interval properties. This allows us to handle long sequences of state transitions in the hardware as they occur when running programs. We demonstrate the feasibility of our approach using the example of an industrial LIN software running on a public domain microprocessor platform.

Categories and Subject Descriptors
B.8.2 [PERFORMANCE AND RELIABILITY]: Performance Analysis and Design Aids; B.6.3 [LOGIC DESIGN]: Design Aids—Verification

General Terms
Verification, Design, Algorithms

Keywords
Formal Verification, Embedded System, Hardware/Software, Abstraction

1. INTRODUCTION

Verification of an embedded system design is a tremendous task and requires a large variety of verification techniques applied at all design levels ranging from the application and system software to the register transfer level (RTL) descriptions of the hardware. In today’s industrial practice, software verification is almost exclusively based on simulation techniques. At the hardware level, formal techniques have gained popularity and complement traditional simulation. In fact, even complex processor implementations can be completely verified formally against their instruction set architecture (ISA) models. It is therefore promising to explore how to extend formal approaches of hardware verification to low-level software such that a formal approach to hardware/software co-verification becomes possible. This opens new perspectives for the use of formal techniques in an industrial setting. Instead of guaranteeing correctness of a processor with respect to the specified ISA the formal guarantee could be given that the application programmer interface (API) behaves as described in a more abstract specification. This would provide a clean and formally verified interface as it is really needed by most users and would be especially useful when the definition of an ISA model is not feasible such as in some cases of highly optimized Application-Specific Instruction Set Processors (ASIPs) and Weakly Programmable IPs [13].

However, formally verifying low-level software, such as bring-up or initialization programs or drivers for bus protocols, in conjunction with the hardware on which they are running is a major challenge. Instead of formally verifying a single processor instruction we now have to verify programs that run over hundreds or thousands of instructions. This complexity can only be handled if we choose formal verification techniques that are best suited to handle models of large complexity. This is the reason why, in the context of this paper, we are particularly interested in formal verification methods by Bounded Model Checking (BMC) [3] or Interval Property Checking (IPC) [15, 14] which are based on satisfiability (SAT) solving and a bounded system model. In particular, the operation-oriented methodology of IPC provides us with the basis for a fairly simple but effective abstraction mechanism.

In IPC, operational behavior of the system is specified within finite time windows using interval properties. Differently from BMC the unrolled circuit is not constrained by the initial states of the circuit. Therefore, the interval properties are verified unboundedly against the system.

In traditional RTL hardware verification behaviors are considered running, typically, over less than a hundred clock cycles. However, in HW/SW co-verification we formulate properties spanning thousands of clock cycles at the hardware level. As a consequence, the computational model becomes intractable even for the most powerful solvers. In this paper, we tackle this problem by an abstraction mechanism that is well adapted to typical situations in HW/SW co-verification and leverages specific strengths of the IPC operation-oriented verification paradigm.

1.1 Related Work

When the size of the system under verification increases its state space grows exponentially. A possible solution to this problem can be to consider more abstract models. The abstract model is conservatively created from the concrete system such that if the property holds on the abstract model, it also holds on the concrete system [5]. Two methods that are commonly used are localization reduction [12] and predicate abstraction such as in [10]. If the abstract model over-approximates the reachable state space a property may fail on the abstract model but still hold on the concrete system. In this case, we need to refine the abstract model. The refinement process is often guided by analyzing false counterexamples [6, 9].
In [4] the authors propose to use CTL properties as a replacement for concrete components in a large system. In [16], the authors extend predicate abstraction by introducing predicates of the transition relations. Similar to [4, 17] we use the properties to abstract the concrete system, however, we do not transfer the properties into automaton models.

Most work on formal software verification such as [2] is based on hardware-independent models. Only a few works exist that consider hardware-dependent software or target formal HW/SW co-verification, such as [7, 8]. However, in those works, the software part still needs to be transferred manually into a hardware-dependent model. Also the resulting models are fairly low-level, thus leading to computational complexity problems when considering larger programs.

### 1.2 Contribution

We propose an abstraction methodology to improve the robustness of interval property checking when proving long, global interval properties of embedded systems. In contrast to other abstraction methods [6, 10, 9], we do not verify the property against the computational model generated from an abstraction system model. Hence, we do not need to calculate the transition relation of some abstract finite state machine. Instead, our abstraction mechanism modifies the unrolled finite state machine of the concrete design by replacing some parts of it by interval properties. Differently from [17] we preserve the concrete transition relation in certain time intervals while abstracting it in others. This is highly useful in HW/SW co-verification where different levels of abstraction are needed in different parts of the program. For example, cycling in a waiting loop may allow a stronger abstraction than executing complex hardware operations. Moreover, our method can abstract the state variables modeling the RAM, hence, a large memory size does not cause the state explosion problem as usually happens when embedded systems are verified.

Our abstraction mechanism allows us to prove meaningful properties against an embedded system consisting of a processor, a RAM module, a ROM module and some peripherals. In such properties, we can specify not only the behavior of the software program as it is executed by the processor (e.g., by referring to program variables) but also the behavior of the system hardware running in parallel with the processor (e.g., by referring to output/input signals of peripherals).

### 2. EMBEDDED SYSTEM VERIFICATION ON THE HARDWARE LEVEL

#### 2.1 Embedded System – Hardware Level

In this paper, we want to verify an embedded system modeled at the hardware level. We consider the hardware and the software program (or program in short) in a joint computational model. This allows us to relate hardware events and corresponding software actions and to detect flaws in their interplay.

Figure 1 shows how to construct the design under verification (DUV) for the embedded system at the hardware level. The program specified at the software level (e.g., in C language) is compiled into executable code and is loaded into the program memory that is connected with the processor on the hardware level (e.g., in an HDL). Therefore, the embedded system on the hardware level consists of a processor connected with a program memory (compiled program loaded in a ROM), a data memory (RAM) and peripherals, e.g., a UART (Universal Asynchronous Receiver/Transmitter). These modules are connected together and are compiled to build the DUV.

The embedded system at the hardware level can be considered as a sequential circuit $C(V, V', X)$ with the set of state variables $V$, the set of next state variables $V'$, and the set of input signals $X$.
Interval Property Checking

Interval Property Checking (IPC) is a SAT-based property checking technique. Similar to Bounded Model Checking, it uses a SAT solver to refute an interval property. An interval property $P$ is an implication $\varphi_1 = (A_1 \Rightarrow C_1)$ where both $A_1$ (called assumption) and $C_1$ (called commitment) are sequence predicates of length $l$. An IPC checker proves that for all sequences $\pi_l = (s_0, \ldots, s_l)$ where the assumption holds on the design the commitment does not hold: $M_l(\pi_l) \Rightarrow (A_1(\pi_l) \Rightarrow C_1(\pi_l))$, i.e.,

$$M_l(\pi_l) \Rightarrow \varphi_1(\pi_l)$$

The property check can be formulated as a SAT problem. Refuting $M_l \Rightarrow \varphi_2$ means satisfying $(M_l \land \neg \varphi_2)$. A satisfying set is a path $\pi_l$ representing a counterexample of the property. It is a false counterexample if the state $s_0$ in the path is unreachable from the initial state. In order to rule out unreachable counterexamples in practice, it is common to add invariants to the proof problem [14]. The strengthened problem looks like this:

$$(\Phi_0(\pi_l) \land M_l(\pi_l)) \Rightarrow \varphi_1(\pi_l)$$

Here, the predicate $\Phi_0(\pi_l)$ is a state predicate characterizing an over-approximation of the reachable state set at the head (i.e., the starting state) $s_0$ of the $l$-sequence $\pi_l$.

2.4 Specification of embedded systems

The behavior of the embedded system is specified on the hardware level using temporal logic. In industrial practice, several languages can be used such as SVA or ITL (InTerval Language), a proprietary language developed by OneSpin Solutions [15]. ITL can be mapped to a subset of LTL as described in the following.

An interval LTL formula is an LTL formula that is built using only the Boolean operators $\land$, $\lor$, $\neg$ and the "next-state" operator $X$. We define a generalized next-state operator $X^k$ that denotes finite nestings of the next-state operator, i.e., if $p$ is an interval LTL formula, then $X^kp = X(X^{k-1}p)$ for $k > 0$ and $X^0p = p$. An interval LTL formula is said to be in time-normal form if the generalized next-state operator $X^k$ is applied only to atomic formulas. Since in LTL, $X(a \lor b) = Xa \lor Xb$ and $X(a \land b) = Xa \land Xb$ and $\neg Xa = \neg X\neg a$, any interval LTL formula can be translated to time-normal form.

It is easy to see how an interval LTL formula can be used to specify an $l$-sequence predicate: The generalized next-state operator refers to the state variables of the system at the different "time" points in the sequence. The maximum power of the $X$ operator occurring in the property determines the length of the property.

Note that in IPC, a property $p$ is proven to hold globally (unboundedly) because the computational model for the proof assumes
an arbitrary starting state $s_0$ (and not the initial state as in BMC). In other words, the property is implicitly preceded by a G operator: $G p = G (a \Rightarrow c)$ with $a$ and $c$ being the interval LTL formula for the assumptions $A_1$ and commitments $C_i$, respectively.

For the embedded system, we normally consider a portion of the system behavior when the processor executes a segment of the program given by an execution sequence $\pi = (s_0, s_1, \ldots, s_k)$. We write the interval property describing the system behavior in the following way:

$$\varphi_t := (PC = \hat{s}_0 \land a) \Rightarrow (X^t PC = s_t \land c)$$

(2)

The assumptions $a$ in this property include definitions to uniquely identify the clock cycle in which the first instruction in the sequence is loaded into the decode stage of the processor, i.e., $(PC = \hat{s}_0 \land a) \Rightarrow \beta(\hat{s}_0)$, as well as constraints to uniquely determine the execution sequence $\hat{\pi}$.

For example, the interval property in Eq. 3 specifies the following behavior of the embedded system: "When the PC is addr1 and there is a start input, then, after 20 clock cycles, the PC will reach addr2 and the value of data (in memory) is increased by 5":

$$\varphi_t = (PC = addr1 \land x = \text{start}) \Rightarrow (X^{20} PC = addr2 \land (X^{20} \text{data}) = \text{data} + 5)$$

(3)

3. PROPERTY-BASED ABSTRACTION

A problem of SAT-based verification techniques such as BMC, or IPC is the complexity of the computational model in Eq. 1. The proof problem $M_L \Rightarrow \varphi_L$ becomes infeasible for the SAT solver for large property lengths $L$. Especially, interval properties for an embedded system are often very long because the operational behavior of the embedded system consists of many program instructions and takes many hardware clock cycles to execute. In this section, we propose a partial abstraction method to reduce the complexity using short properties.

3.1 Motivation and Basic Idea

When the processor executes the program it performs many local execution steps and uses many temporary registers. This helps to increase the performance and the usage of hardware resources in the processor. For example, to calculate the statement $c = a + b$, the processor needs to perform a number of steps: load the values of memory cells $a, b$ into temporary registers R1 and R2; add R1 and R2 and put the result into R3; and finally store R3 into memory cell c. Moreover, each of the above steps can be divided further into shorter steps in the pipeline structure of the processor. The sequence of such steps corresponds to a long run of the system. Therefore, the iterative circuit model representing the run consists of many logic components and is complex.

However, the results of local steps are usually not needed when the statement is considered in a global operation of the system. Only the final result of the statement needs to be specified. Hence, the behavior of a statement, or a portion of the program can be specified by an interval property that refers to the state variables at a few time points, only. For example, the property in Equation 3 refers to the PC register, the input x and the data register at the time points 1 and 20. The translation of the property, thus, consists of less logic and is simpler than the iterative circuit model. Consequently, if we replace the iterative circuit model corresponding to a program portion by the translation of the property specifying that program portion we can obtain a much simpler computational model for proving a global software operation. The basic idea of our abstraction method is illustrated in Figures 3 and 4.

The overall goal (Fig. 3) is to prove a “long” property $\varphi_L$ spanning many clock cycles. The straightforward approach of constructing a long unrolling of the circuit into L timeframes and attempting to prove the property directly fails because of complexity. However, a number of smaller properties $\varphi^1, \varphi^2, \varphi^3$ are feasible for the SAT solver to prove. These properties are now used to partially or fully abstract the circuit unrolling. Wherever a short property $\varphi^k$ describes a sub-sequence of the instructions verified by the long property $\varphi_L$ it replaces the corresponding time frames of the transition function of the embedded system’s hardware.

Note that this abstraction is not a trivial one as may seem at first glance. We are concatenating the short properties not in an implicative manner, i.e., we do not prove that the commitment $c_i$ of a property $\varphi^i$ implies the assumption $a_{i+1}$ of the successor property $\varphi^{i+1}$ nor would we make use of this implication should it exist. (This would correspond to some horizontal implication arrows in Fig. 4.) In fact, it may even be that properties are concatenated with original time frames of the hardware transition function if these time frames cannot be covered by a proven shorter property. We do, however, exploit transitivity of the vertical implication relationships indicated in Fig. 4: If time frames can be partially abstracted by proven short properties and the long property can be proven based on this abstraction then the long property also holds for the unabstracted circuit unrolling.

Note that our abstraction mechanism allows modular and hierarchical abstraction. Modularly, we can replace the iterative circuit model of some components in the system by the properties that are proven for those components. Hierarchically, we can also replace partially abstracted circuit unrollings by global properties to prove even longer properties.

In the next section, we will describe our abstraction method in more detail.

3.2 Property-Based Abstraction Method

We begin with a set of $m$ properties $\{\varphi^k_l\}$ with $1 \leq k \leq m$. In this notation, superscript $k$ is an index distinguishing properties in the set, subscript $l_k$ denotes the length of each property $k$. We assume that every property has been proven to hold on the design, i.e., $M_{l_k} \Rightarrow \varphi^k_{l_k}$.

The goal is to prove a “long” property $\varphi_L$ with a large length $L$. Suppose the computational model for a straightforward IPC check, $M_L \Rightarrow \varphi_L$, is too complex for the SAT solver to handle. However, a simplified model abstracted using shorter properties may be feasible. For example, assume property $\varphi^2_{L2}$ has been proven, already. If $L = l_1 + l_2$, i.e., $M_L = M_{l_1} \circ M_{l_2}$, the SAT solver may be able to prove $M_{l_1} \circ \varphi^2_{L2} \Rightarrow \varphi_L$, implying that $M_{l_1} \circ M_{l_2} \Rightarrow \varphi_L$ holds, also.

In general, it is desirable to abstract as many time frames as possible by properties in order to reduce the proof complexity. The properties should be chosen such that they specify program execution sub-sequences of the longer property we want to prove. This is achieved by including $\beta$-mappings into the assumptions and commitments to describe the starting and ending program positions of the considered execution sub-sequences. We proceed in the following way.

Step 1. If the proof problem $M_L \Rightarrow \varphi_L$ is feasible for the IPC checker, prove the property $\varphi_L$ and add it to the set of proven properties.

Step 2. If the proof does not succeed, simplify the computational model as follows.

a. For a proven property $\varphi^k_{l_k}$, analyze whether the execution sequence $\pi_{l_k} = (s_0, s_1, \ldots, s_{l_k})$ described by the property is a
subsequence of the execution sequence \( \pi_L = (t_0, t_1, \ldots, t_L) \) of the long property \( \varphi_L \), i.e., there exists a \( j \) such that \( t_j = s_0 \) and \( \pi_L = (t_0, t_1, \ldots, t_{j-1}, s_0, \ldots, s_t, t_{j+1}, \ldots, t_L) \). If so, replace the time frames of the unrolled design corresponding to the subsequence by the property. The proof problem based on this abstracted circuit model becomes

\[
M_j \circ \varphi^\sharp_L \circ M_{(L-1)-t_k} \Rightarrow \varphi_L \tag{4}
\]

b. Repeat this for all proven properties \( \{\varphi^\sharp_L\} \).

Step 3. If the IPC checker is able to prove Eq. 4, add the property \( \varphi_L \) to the set of proven properties.

Step 4. If the IPC checker is not able to prove Eq. 4, we need to partition the execution sequence \( \pi_L = (t_0, t_1, \ldots, t_L) \) into shorter sub-sequences. For the sub-sequence starting from \( s_0 \), we verify the system behavior for this sub-sequence in a new property. Going back to step 2, this new property is then used to simplify the model.

In our approach, steps 1, 2 and 3 can be done automatically. Only in step 4, the interaction of the verification engineer is needed to construct the new property. For this property, the starting and ending CFG states \( s_0 \) and \( t_k \) are chosen such that the length \( l \) of the sequence \( \pi_L \) is as long as possible. However, the engineer needs to specify the assumption \( a \) and the commitment \( c \) in the property.

The correctness of our abstraction method is guaranteed by the following theorem.

**THEOREM 1.** Given a “long” property \( \varphi_L \) and a “short” property \( \varphi_L \). The short property is known to hold on the design, i.e., \( M_l \Rightarrow \varphi_L \) is true. If \( M_{L-1} \circ \varphi_L \Rightarrow \varphi_L \) holds then do so \( M_L \Rightarrow \varphi_L \), i.e., also the long property holds.

**PROOF 1.** Because \( M_l \Rightarrow \varphi_L \), also the implication shifted to the right by \( L-1 \) clock cycles is true: \( \text{next}(M_{l-1} \circ L-L) \Rightarrow \text{next}(\varphi_L \circ L-L) \). We conjoin the left-hand side with \( M_{l-1} \circ \text{next}(M_{l-1} \circ L-L) \Rightarrow \text{next}(\varphi_L \circ L-L) \) and, finally, \( M_{l-1} \circ \text{next}(M_{l-1} \circ L-L) \Rightarrow M_L \circ \text{next}(\varphi_L \circ L-L) \). This is equivalent to \( M_{L-1} \circ M_l \Rightarrow M_L \circ \varphi_L \) or \( M_L \Rightarrow M_L \circ \varphi_L \). If, as required by the theorem, \( M_{L-1} \circ \varphi_L \Rightarrow \varphi_L \) holds then, by transitivity of implication, also \( M_L \Rightarrow \varphi_L \) holds.

To the best of our knowledge, this is the first time that the abstract computational model for proving properties is constructed from a combination of both, instances of the abstract transition functions as given by the properties and instances of the concrete transition function.

4. EXPERIMENTAL CASE STUDIES

4.1 Experimental Setup

The proposed techniques were evaluated on an embedded system consisting of an open source 32-bit processor Aquarius [11] being connected with two memory modules, a RAM and a ROM, and a peripheral module, a UART. These hardware components are described in 6850 lines of Verilog code. The processor executes a program in C language which is compiled and loaded into the ROM module. As a result, the design under verification (DUV) at the hardware level consists of the processor, the peripheral and the memory component that contains the software program. These components are completely specified in a hardware-level language and can be verified against properties by the IPC checker.

Although the properties are formulated in a hardware-level language it is possible to refer to program positions and variables in the software since the required information is also translated into the hardware level using a simple converter tool. In the next two sections, we describe two experiments to prove the behavior of the system where the processor executes two programs. The first experiment handles the system with the Fibonacci program and the second experiment considers the system with a LIN device driver developed by Infineon Technologies, Inc. LIN (Local Interconnect Network) is a sequential network protocol commonly used in automotive systems.

4.2 Fibonacci Program

In this experiment, we compare the performance and the robustness of the proposed method against a brute-force approach without abstraction.

We verified the Fibonacci program shown in Figure 2. We specified an interval property describing a piece of system behavior where \( i \) Fibonacci numbers are calculated. We proved the property against the concrete system using the commercial IPC checker from OneSpin Solutions [15].

In Table 1, we report the CPU time and the memory consumption that the IPC checker needed to prove the properties in columns 3 and 5, respectively. Columns 1 and 2 show the number of Fibonacci numbers \( i \) that are calculated and the length of the corresponding properties. As we can see in the table, without the proposed abstraction technique the IPC checker reports memory-out (marked “MO” in the table) when it was given the circuit unrolled over 718 clock cycles to prove the property specifying that 35 Fibonacci numbers had been calculated.

Applying the proposed method, we consider the original properties as long properties and we formulate short properties specifying the behavior of the system when the loop body of the program is executed as described in Section 3.2. Then we use these properties to abstract the unrolled iterative circuit model when proving the long properties. The CPU time and memory consumption are shown in columns 4 and 6 of Table 1. Note that the proposed technique can prove even significantly longer properties depending on what aborting conditions we choose.

<table>
<thead>
<tr>
<th>( i )</th>
<th>length</th>
<th>CPU time (s)</th>
<th>Memory usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>118</td>
<td>99</td>
<td>6137</td>
</tr>
<tr>
<td>6</td>
<td>138</td>
<td>114</td>
<td>6502</td>
</tr>
<tr>
<td>7</td>
<td>158</td>
<td>142</td>
<td>6907</td>
</tr>
<tr>
<td>8</td>
<td>178</td>
<td>164</td>
<td>7772</td>
</tr>
<tr>
<td>9</td>
<td>198</td>
<td>197</td>
<td>8662</td>
</tr>
<tr>
<td>15</td>
<td>318</td>
<td>390</td>
<td>14038</td>
</tr>
<tr>
<td>20</td>
<td>418</td>
<td>644</td>
<td>18894</td>
</tr>
<tr>
<td>25</td>
<td>518</td>
<td>768</td>
<td>23447</td>
</tr>
<tr>
<td>30</td>
<td>618</td>
<td>914</td>
<td>26665</td>
</tr>
<tr>
<td>35</td>
<td>718</td>
<td>729</td>
<td>MO</td>
</tr>
<tr>
<td>40</td>
<td>818</td>
<td>–</td>
<td>445</td>
</tr>
<tr>
<td>45</td>
<td>918</td>
<td>–</td>
<td>517</td>
</tr>
<tr>
<td>50</td>
<td>1018</td>
<td>–</td>
<td>658</td>
</tr>
</tbody>
</table>

Table 1: Verifying the Fibonacci program

4.3 LIN driver

In this experiment, we applied the proposed method to an industrial software implementing a LIN master node. The software has

<table>
<thead>
<tr>
<th>Property (name)</th>
<th>Length (cycles)</th>
<th>Proof size (AND gates)</th>
<th>CPU time (s)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>first_new_frame</td>
<td>110</td>
<td>567/981</td>
<td>648</td>
<td>10,722</td>
</tr>
<tr>
<td>wait_synchbreak</td>
<td>30</td>
<td>155,961</td>
<td>68</td>
<td>2,168</td>
</tr>
<tr>
<td>load_synchbreak</td>
<td>920</td>
<td>199,697</td>
<td>625</td>
<td>15,949</td>
</tr>
<tr>
<td>start_bit_synchbreak</td>
<td>810</td>
<td>181,604</td>
<td>474</td>
<td>13,967</td>
</tr>
<tr>
<td>bit1_synchbreak</td>
<td>810</td>
<td>181,604</td>
<td>474</td>
<td>13,635</td>
</tr>
<tr>
<td>bit2_synchbreak</td>
<td>810</td>
<td>181,604</td>
<td>517</td>
<td>13,969</td>
</tr>
<tr>
<td>bit3_synchbreak</td>
<td>810</td>
<td>181,604</td>
<td>472</td>
<td>14,008</td>
</tr>
<tr>
<td>bit4_synchbreak</td>
<td>810</td>
<td>181,604</td>
<td>473</td>
<td>14,010</td>
</tr>
<tr>
<td>bit5_synchbreak</td>
<td>810</td>
<td>181,605</td>
<td>480</td>
<td>13,967</td>
</tr>
<tr>
<td>bit6_synchbreak</td>
<td>810</td>
<td>181,605</td>
<td>476</td>
<td>14,003</td>
</tr>
<tr>
<td>bit7_synchbreak</td>
<td>810</td>
<td>181,604</td>
<td>490</td>
<td>13,969</td>
</tr>
<tr>
<td>bit8_synchbreak</td>
<td>810</td>
<td>181,602</td>
<td>475</td>
<td>13,964</td>
</tr>
<tr>
<td>bit9_synchbreak</td>
<td>810</td>
<td>181,597</td>
<td>481</td>
<td>14,014</td>
</tr>
</tbody>
</table>

Table 2: Verifying a LIN implementation
about 850 lines of C code. It was ported to the open-core-based platform described above.

Our goal is to verify that the HW/SW system correctly implements all LIN transactions. As an example, we proved that the synchbreak symbol is first sent when the LIN node wants to start a LIN message transmission. Because symbols in the LIN frame are transmitted as serial bytes at a very slow rate with respect to the clock rate of the processor the length of the considered behavior in terms of the processor clock cycles is often very large. For example, at the Baud rate 38400 bps, one bit in the synchbreak symbol needs to be transmitted in \( T_{\text{field}} = 1.5/38400 \) seconds. If the clock frequency of the processor is 20 MHz, the length of the property specifying the transmission of 1 bit is \( k = T_{\text{field}} \cdot 20 \cdot 10^6 \text{Hz} = 781 \) cycles. Consequently, the computational model to prove such properties becomes very complex and cannot be handled by IPC.

We applied the proposed method to prove properties stating that each bit of the synchbreak symbol is transmitted correctly to the LIN bus. In the first step, we constructed two short properties specifying the behavior of the software executed in the processor. The first property, called first_new_frame, specifies that the synchbreak symbol is written to the UART correctly. The second short property, called wait_synchbreak, specifies that while the UART is sending synchbreak to the LIN bus a software loop is executed to wait for an interrupt from the UART signaling that it finishes the transmission. In the second step, we used these two properties to abstract the computational model to prove the long properties stating that individual bits are sent correctly to the LIN bus (UART output signal TxD). To do this, one instance of first_new_frame and about 20 instances of wait_synchbreak are used in the abstraction. The experimental results are shown in Table 2. In the table, columns 1 and 2 show the properties and the corresponding lengths. Column 3 shows the size of the proof problem in number of AND gates. Columns 4 and 5 show the CPU time and the memory usage that were needed to prove the properties. In the table, all properties besides the first two are long properties that were proven using the described abstraction. Note that without abstraction IPC runs out of memory in all cases when trying to prove a long property. (This is not explicitly stated in the table.)

In this way, we proved that all LIN transactions work correctly in the implemented system. We are not aware of any property checking tools or techniques that could be successfully applied to this HW/SW co-verification problem. Existing model checkers would consider the software in a hardware-independent way and would possibly miss bugs that occur in the hardware or the interplay between hardware and software. The proposed approach, however, can complement HW/SW co-simulation techniques and has potential to provide complete correctness proofs for low-level drivers in embedded systems ensuring the correctness of hardware, software and their interaction.

5. CONCLUSION

We propose a novel method to prove long interval properties in HW/SW systems. It is based on an operation-oriented verification paradigm that is employed successfully in industry for hardware. We propose a property-based abstraction scheme and show that this can increase the length of provable interval properties by an order of magnitude. In future work, we intend to combine this approach with path predicate abstraction as proposed in [18] to create provably correct system abstractions for large HW/SW-systems.

6. REFERENCES


