IPSJ Transactions on System LSI Design Methodology Vol.6 135–145 (Aug. 2013)

[DOI: 10.2197/ipsjtsldm.6.135]

Regular Paper

A New Formal Verification Approach for Hardware-dependent Embedded System Software

Bernard Schmidt[1,a)] Carlos Villarraga[1] Thomas Fehmel[1] Jörg Bormann Markus Wedler
Minh Nguyen[2] Dominik Stoffel[1] Wolfgang Kunz[1]

Received: February 8, 2013, Revised: April 19, 2013,
Accepted: May 11, 2013, Released: August 5, 2013

Abstract: This paper describes a method to generate a computational model for formal verification of hardware-dependent software in embedded systems. The computational model of the combined HW/SW system is a *program netlist* (PN) consisting of *instruction cells* connected in a directed acyclic graph that compactly represents all execution paths of the software. The model can be easily integrated into SAT-based verification environments such as those based on Bounded Model Checking (BMC). The proposed construction of the model allows for an efficient reasoning of the SAT solver over entire execution paths. Program netlists are compositional. The paper presents how they can be combined to model interrupt-driven systems. We demonstrate the efficiency of our approach by presenting experimental results from the formal verification of an industrial LIN (Local Interconnect Network) bus node, implemented as a software driver on a 32-bit RISC machine.

Keywords: low-level software, interrupt-driven software, HW/SW co-verification, property checking

1. Introduction

Formal approaches based on property checking are enjoying increasing popularity when verifying the functional correctness of Systems-on-Chip (SoCs). More and more advanced commercial technology becomes available that provides solutions for formally verifying the hardware of an SoC. However, there is a growing need to also incorporate the low-level software into the formal verification methodology. In state-of-the-art embedded systems, low-level system software is tightly coupled with the hardware structures on which it is running. Separate proofs of hardware and software correctness are often not sufficient to assess the overall behavior of the embedded system. This problem increases as the borderline between hardware and software is becoming blurred in highly optimized application-specific designs where the semantics of a program are sometimes only defined in a specific hardware context [1]. In this paper, we propose a computational model that can be used to integrate low-level software verification into hardware verification environments, thus allowing for verification of low-level software in its particular hardware environment.

1.1 Comparison with Related Work

Formal verification of software is a field that has been studied extensively over several decades. A good overview on basic approaches, especially as they are relevant in the context of embedded system design, is provided in Refs. [2] and [3].

Typically in those approaches, the software is described by “simple programs” as some kind of (finite) state transitions system that are processed by model checking and related techniques. Programs are given in high-level languages like C. Methods adopting an unbounded paradigm [4], [5], [6], [7], [8], [9], [10] and a bounded paradigm [11], [12], [13] can be distinguished. Differences between tools and techniques result from the underlying proof methods (enumerative (stateful, stateless), symbolic) and the employed abstraction techniques (iterative abstraction-refinement based on localization reduction and/or predicate abstraction).

Most characteristics which differentiate the approach proposed in this paper from previous ones result from our objective to verify *hardware-dependent* low-level software. In hardware-dependent software verification the behavior of a program is examined with respect to its impact on the concrete hardware on which it is running. This is important when verifying the detailed behavior of the hardware/software interface in an embedded system. For example, an SoC bus system can be composed from hardware and software components. In order to verify protocol compliance of the bus we need to analyze the detailed behavior of the software with respect to the signals of the concrete hardware. The model we propose here therefore contains objects that directly represent signals of the hardware architecture. The properties that can be verified in our model may relate software behaviour to concrete hardware signals. The same hardware signals can be referenced in standard property checking applied to the surrounding hardware so that both, the software and the hardware behaviour of the system, can be covered without gaps.

Most software verification techniques reported in literature, on

[1] University of Kaiserslautern, Kaiserslautern, Germany
[2] Hanoi University of Science and Technology, Hanoi, Vietnam
[a)] bschmidt@eit.uni-kl.de

the other hand, are *hardware-independent*. The semantics of the program is defined by a high-level language and the program behavior is examined independently of the target hardware platform. The proposed approach leverages many elements of existing software and hardware verification techniques. Its novelty results from how these elements are combined to create a computational model that is adequate for verifying hardware-dependent software. This will be discussed in the following paragraphs.

A straightforward approach to hardware-dependent software verification can be based on Bounded Model Checking (BMC) [14]. A Register Transfer Level (RTL) description of the CPU and its memory with the considered program is unrolled for a finite number of steps that is determined by the maximum number of clock cycles along the program's longest execution path [15], [16]. This immediately creates a model that represents the entire program semantics in terms of the underlying hardware. Obviously, such a brute-force approach is not tractable for HW/SW systems of realistic complexity. For this reason, a specific abstraction mechanism for this scenario was proposed in Ref. [16].

In spite of its complexity, such a hardware-style BMC approach is actually attractive for hardware-dependent software verification since the hardware can be easily represented at the desired level of detail. For example, we may represent the hardware by the concrete RTL implementations or, depending on our verification objectives, by an abstract CPU model at the Instruction Set Architecture (ISA) level. Since we wish to have a hardware-dependent view on the program it is an attractive feature of this approach that the entire behavior of the program is represented *implicitly* by the unrolled hardware. Note that the complexity issues related to this approach do not only result from the sheer size of the unrolled model. A main drawback of this approach arises from the fact that not only the computational semantics of the individual program steps but also the entire *control flow* of the program is represented only implicitly by the hardware. A BMC approach formulated like this does not have any explicit view on the actual execution paths of a program. This makes SAT reasoning on such a model inefficient since the knowledge on possible execution paths must be learned in a tedious way through backtracking, clause learning and related concepts.

In contrast, software verification techniques usually follow a path-oriented approach where paths are traced and represented in an *explicit* way. This is, for example, the case in approaches based on Symbolic Execution, such as Refs. [17], [18]. Symbolic execution enumerates program execution paths and checks conditions for verification along these paths by specialized verification algorithms. Symbolic formulas are created that explicitly represent all possible input scenarios along the considered paths.

While an explicit representation of the program's control flow is certainly a strong advantage of this approach it can be considered a disadvantage that also the program's computation is represented by formulas, which are explicitly generated. Especially in the context of hardware-dependent verification these formulas and the specialized verification algorithms may become overly complex. When compared to the approach proposed here, it may be considered a disadvantage that the evaluation of the program flow is mingled with reasoning to prove the considered properties. In a hardware-dependent environment, the proof goal may need to be examined in combination with a large number of paths, leading to an explosion of the symbolic formulas.

In CBMC [13] a different hardware-independent approach is taken where conventional BMC is used to build a SAT formula for the program computation based on unrolling the control flow graph (CFG) of a C program. This combines some of the advantages of BMC with a path oriented view on the program. Note, however, that the SAT formulas built by CBMC operate directly at the C level. The semantics of the C statements is directly encoded into the generated model. This is adequate for hardware-independent software verification, as intended by CBMC, but becomes an obstacle when verifying hardware-dependent software based on machine instructions. Rather than directly operating on hardware descriptions, as proposed by our approach, verification by CBMC or by other hardware-independent verification methods would require to model the machine program as well as the surrounding hardware in a high level software language.

1.2 Basic Idea

Taking all of this into account, clearly, in hardware-dependent software verification the *control flow* and the *computation* of a program need to be considered separately and have to be represented and processed in different ways. Therefore, this paper proposes a computational model that splits the overall verification tasks into two phases.

In the first phase, the program semantics is only considered to the extent needed to evaluate its control flow. We generate an execution graph that explicitly represents the control flow, and apply pruning and merging techniques automatically. This leads to an *explicit* representation of the *control flow*, as in Symbolic Execution. However, the program *computation*, as it is evaluated for solving verification tasks, is completely ignored in this phase, and no symbolic formulas to represent the program computation are generated at all. The program's computation is instead represented in an *implicit* way by hardware models. This is similar as in hardware-dependent BMC approaches. Thus, for our hardware-dependent software verification approach we combine an implicit representation of the program computation by hardware with the advantages of a path oriented representation of the control flow as it is used by most hardware-independent verification methods. Technically, in order to make the control flow explicit, a special control structure will be incorporated into our hardware-dependent computational model.

The resulting model is called *program netlist*. It contains all relevant information for verifying the functional correctness of the software including the communication details required for interfacing with the hardware environment. For example, when the hardware environment interfaces with the CPU using interrupt requests. Our hardware-dependent model contains signals that model interrupt events and their priorization. This allows us to verify the functional behaviour of the interrupt service routines in conjunction with the hardware-implemented interrupt mechanisms. Correctness of the context switch and the involved priorization mechanisms are examples of proofs which can be per-

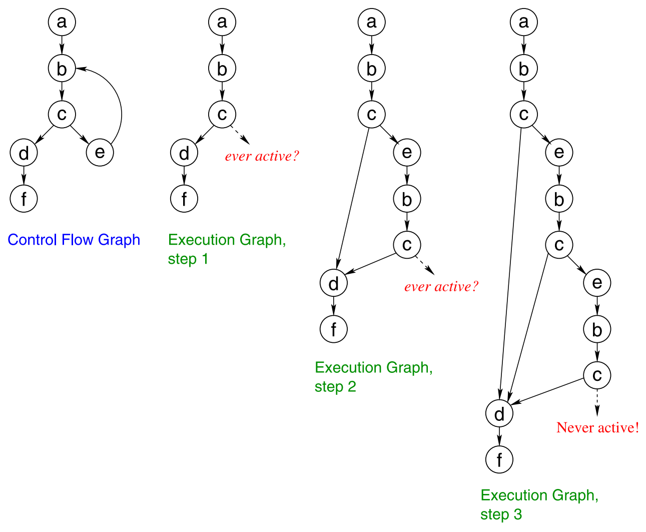

Fig. 1 Unrolling a CFG into an execution graph.

formed on the proposed model.

We can also exploit its compositionality when modeling interrupt-driven systems, as will be described in Section 3.

In the second phase of our approach, the actual verification task is solved by applying a standard property checker to the generated model. Compared to a brute-force BMC approach, a SAT solver will greatly benefit from the explicit control flow representation in this model. In contrast to Symbolic Execution approaches, it is now left to the intelligence of the SAT solver to enumerate *paths* and to explore the *hardware-represented program computation* only to the extent needed for proving a given property.

A *program netlist* models executions of programs that are finite. Assuming finite paths is valid, especially for low-level embedded software such as interrupt-based driver services or real-time tasks. The basic mechanism for generating a program netlist is to first transform a control flow graph into an intermediate data structure that models the finite execution paths of the software. The control flow graph of the software is iteratively "unrolled" at branching nodes. The unrolling is controlled by auxiliary property checks on intermediate versions of the generated model.

Figure 1 illustrates this process. The control flow graph shown on the left is unrolled node by node into the execution graph shown on its right, until branching node c is reached. We then perform two property checks, one for each branch, on the intermediate model corresponding to the shown execution graph. The properties check whether there can be an execution run of the software such that this branch is taken. If the branch condition depends on an external input with arbitrary values both branches are determined to be active and will be further unrolled.

In our example, the left branch fulfills this condition, and we continue unrolling the nodes d and f. The right branch of node c is checked in the same way. Also for this branch there exists an execution. The second graph displays the situation after continuing the unrolling with nodes e, b and c. Again, branching node c is visited. The left branch can be proven to be taken in some program execution. However, instead of creating a new copy of node d, we reuse the existing instance of d as shown. This process, called "merging," is important to keep the model compact.

In our example, also the right branch of the second node c can be shown to become active in some program run. The third execution graph on the right shows the result after one further unrolling of the loop. This time the right-branch condition at node c is never fulfilled, hence there is no further unrolling here.

The resulting data structure is a directed acyclic graph representing explicitly all possible finite execution paths of the program. Importantly, each node in the execution graph on the right side of Fig. 1 can be traversed at most once by a specific program run and corresponds to a well-determined computational step in a specific program location. This together with the finiteness of the model is exploited to create a program netlist where each node in the execution graph of Fig. 1 is instantiated with a description of a single hardware instruction executed in that step.

Note that the explicit representation of the program's control flow, as expressed by the execution graph, allows us to consider, at each of its nodes, the system under the constraint that only one specific instruction is executed by the CPU. This drastically simplifies the logic required for describing the combined hardware/software behavior.

We will show in Section 2 how this explicit view on the control flow, as it is common in hardware-independent software verification, is combined with an implicit hardware representation of the program computation, like in standard BMC for hardware.

For representing interrupt-driven low-level software architectures we propose a procedure in which the computational models corresponding to each software component are first generated independently and then combined into a global representation. This relies on the fact that program netlists are compositional. For an efficient composition we propose a simplification scheme that reduces the complexity of the final model drastically. Reference [19] presents a composition scheme that is based on related concepts for simplifications. However, their approach differs from ours as it is based on an explicit model checking paradigm in which software components are not handled independently during model generation.

The remainder of the paper is organized as follows. Section 2 introduces the proposed hardware-dependent, bounded computational model for low-level software verification. We show how to incorporate the circuit description into a program netlist that is automatically generated from the control flow graph of the program. The discussion begins with developing the basic model and then describes how we reduce its complexity by merging (Section 2.1) and by pruning of dead branches (Section 2.2). The modeling of interrupts using program netlists is presented in Section 3. Finally, in Section 4 we report on two case studies evaluating the proposed model.

2. Hardware-dependent Software Model

The following analysis is based on finite unrollings of the transition logic of an FSM model of the processor and its program. We show how the elements of the program netlists can be viewed as the results of logic optimization operations such as logic duplication, relocation of multiplexers, simplification by constant propagation and node merging. In this view, it becomes evident why the constructed model is both, compact, and correct.

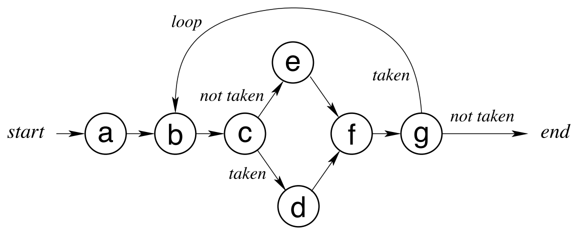

Fig. 2 Example of a control flow graph.

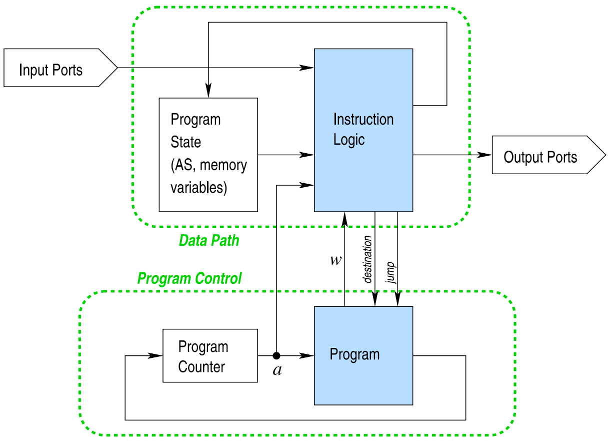

Fig. 3 Abstract HW/SW model.

We model a machine program based on a simple definition of a *control flow graph* (CFG) as a graph $G = (V, E)$ with a set of nodes V and a set of edges $E \subseteq V \times V$. While usually the nodes of the CFG are interpreted as basic blocks, in our case, they represent *machine instructions*. A machine instruction C is a pair $C = (a, w)$ of an instruction address a and the instruction word w stored at that address in the program memory. There is an edge (v_i, v_j) between two instructions v_i and v_j if the program can make a transition from v_i to v_j.

Figure 2 shows an example of a control flow graph of a machine program. The control flow graph can also be understood as an abstract finite state machine controlling the combined hardware-software system. The nodes of the graph correspond to its control states.

The basis of our analysis is a finite state machine model of the combined hardware and software as shown in **Fig. 3**. The machine program is stored in the program memory. For simplicity, we assume that the program location is fixed and known a priori. Extensions of the methodology to code relocation as it happens during linkage and loading are straightforward and will not be discussed here. In our abstract HW/SW model, there is a finite state machine called *program control* consisting of the machine program and the program counter (PC) of the processor. The inputs to this state machine are a jump command and a jump destination. The outputs of the state machine are the PC value and the instruction word w of the machine instruction at the memory address a pointed to by the PC.

Figure 4 shows the program control FSM of the model in more detail.

The remainder of the model is a finite state machine (called *datapath*, cf. Fig. 3) consisting of the *program state*, *input ports* and *output ports* and *instruction logic*. The program state (PS)

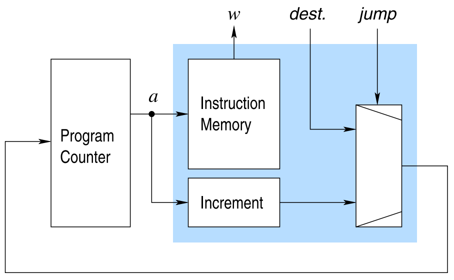

Fig. 4 Program control FSM.

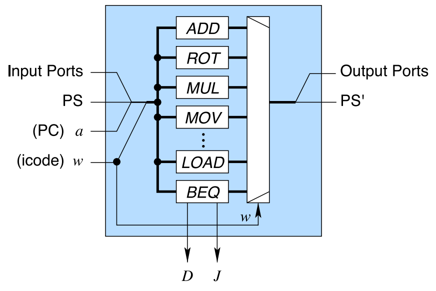

Fig. 5 Instruction logic block.

can be further divided into the variables of the software in data memory, and the *architectural state* (AS) comprising the programmer-visible processor registers. The input ports are locations in the system's data memory space that are written by the environment (e.g., by some peripherals) and read by the program. Likewise, the output ports are data locations where the program writes its results. This interchange of information with the environment can be carried out using independent or shared communication channels as in memory-mapped I/O systems. The block "instruction logic" as shown in **Fig. 5** contains a description of the programmable hardware at a level of abstraction that is appropriate for the overall verification task. For example, it can be an RTL description, a system-level description of a virtual prototype, or an Instruction Set Architecture (ISA) level description of a processor. As will be shown, our method will always instantiate this hardware description restricted to a specific instruction. Therefore, in a pre-processing phase, we derive an "instruction cell" for each processor instruction by restricting and simplifying the hardware description for the considered instruction.

The signal w represents the instruction word located at the program address a. It is used to select the instruction cell corresponding to the current program location (PC) and to multiplex its results back to the program state and/or to the output ports of the model. Control flow machine instructions as, for example, the conditional branch instruction *BEQ* in Fig. 5, are special in that they produce two signals, jump J and destination D, that are both fed back to program control.

In this abstract model, both finite state machines – program control and data path – are synchronized with the same clock signal. One tick on this abstract clock corresponds to the execution of a single instruction in the program.

The proposed model for hardware-dependent software is now obtained by a special unrolling of this abstract model driven by the CFG of the program. In standard Bounded Model Checking (BMC) [14] we would unroll the abstract model for a num-

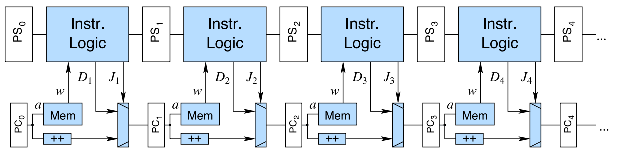
Fig. 6 Unrolling the abstract model. PS denotes program state, PC denotes program counter. Input and output ports are not shown.

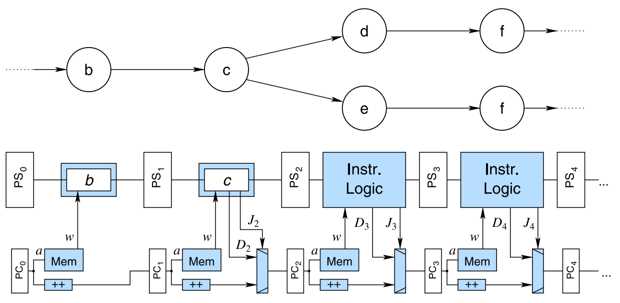
Fig. 7 Unrolled abstract model with conditional jump.

ber of time frames starting from some instruction in the machine program (**Fig. 6**). By "time frame," in the following, we mean one instance of the transition logic of the considered finite state machine. In this unrolling, the program counter is uniquely determined until there is a conditional jump. When the program counter in a time frame is a constant value then also the instruction word is constant. The instruction logic can be simplified by constant propagation so that it only contains the instruction cell corresponding to the instruction word.

In the unrolling, the select signal named *jump* of the branch multiplexer in Fig. 4 is fed with logic '0' for non-branching instructions, logic '1' for unconditional jumps and with some logic function of the registers for conditional jumps. Note, however, that already at the first conditional jump in the unrolling the output of the multiplexer yielding the new PC value is no longer a constant function, i.e., no constants can be propagated and the time frames following the conditional jump contain the full instruction logic as well as the instruction memory of the abstract processor model of Figs. 3 and 4.

Figure 7 shows an example of unrolling the programmed hardware between control states b and f of the CFG of Fig. 2. The first instruction (b) is a non-branching instruction, the next instruction (c) is a branching instruction. Obviously, already the values of the program counter PC_2 and the program state PS_2 are no longer uniquely determined and hence no logic simplification by constant propagation is possible for the third and all subsequent time frames. (For reference, the upper part of Fig. 7 shows the control flow states modeled by the the unrolling. For example, the third time frame models two possible instructions, (d) and (e), at two program locations.)

However, consider the multiplexer accounting for the conditional jump. We can move this multiplexer across the blocks connected to its output, creating copies of these blocks in each of its inputs. This duplicates the logic; however, since the PC value is then again uniquely determined in each of the copies the propagation of constants can continue, significantly simplifying the logic. **Figure 8** shows the results of moving the multiplexer of the first conditional jump to the end of the unrolling. The upper thread

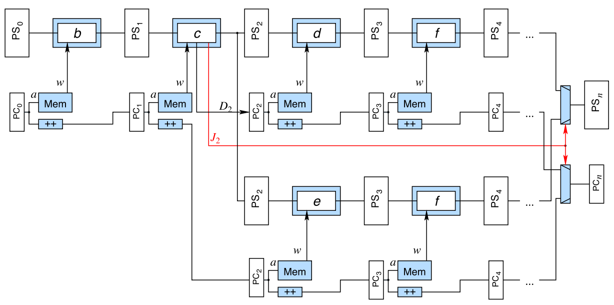
Fig. 8 Duplicating logic by moving a jump multiplexer.

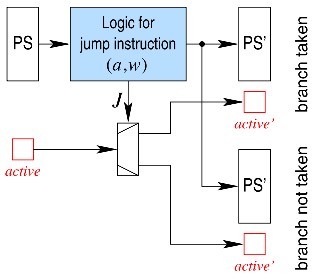

Fig. 9 Instruction cell for branching.

of time frames shows the case for the select signal $J_2 = 1$, i.e., a jump to destination D_2 is taken at $t = 2$. The lower thread shows the case where the jump is not taken, i.e., $J_2 = 0$ and the program counter is simply incremented.

Of course, every relocation of a jump multiplexer doubles the logic at the output of the multiplexer. We could imagine moving all jump multiplexers of a given unrolling to the end. This would create a tree structure with as many leaves as there are execution paths between the first and last instruction of the unrolling. Each duplicated time frame can be simplified by constant propagation so that its instruction logic block contains only the instruction cell for a single instruction. The resulting logic tree is a combinational circuit representing all execution paths through the unrolled machine program. Each time frame represents one instance of a machine instruction $C = (a, w)$ with the instruction word w stored at program memory address a. The size of the execution tree is exponential in the number of conditional jumps within the program. Fortunately, by *merging* of nodes, this tree can be turned into a more compact, directed acyclic graph. We will discuss how and when to do that shortly. The advantage of this construction has already been discussed in Section 1: it exposes the program's control flow in the hardware-dependent model. Execution path fragments between branching and merging points are represented explicitly. This is in contrast to a simple BMC unrolling of the transition relation which represents all execution paths in a single chain of time frames. The resulting structure is the program netlist, our proposed computational model for verification.

The program netlist is constructed from three types of components: instruction cells for branching instructions, instruction cells for non-branching instructions and so-called merge cells. **Figure 9** shows an instruction cell for branching instructions. The instruction logic block drives the jump signal, J. Instead of routing J directly to the multiplexer that was relocated (as in Fig. 8) we instead carry a bit-signal called "active" along with each thread of time frames. This signal is generated from a de-

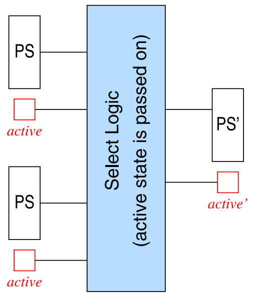

Fig. 10 "Merge cell" for recombining execution paths.

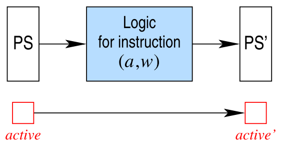

Fig. 11 Instruction cell for non-branching instructions.

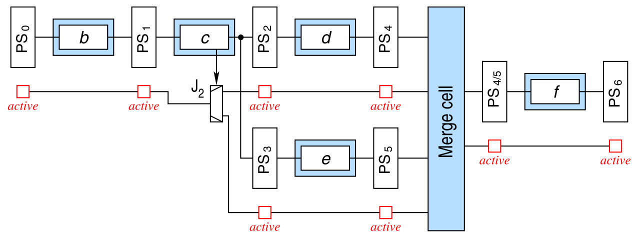

Fig. 12 Program netlist after merging.

multiplexer controlled by J. It is asserted for the taken branch and de-asserted for the non-taken branch for a given program run. The functionality of the relocated multiplexer is implemented by a merge cell (**Fig. 10**). This cell simply passes the values of the program state PS that is marked with the active bit to its outputs. This construction is functionally equivalent to the original one with multiplexers. It simplifies building the program netlist from modular components and it allows identification of "dead branches" as will be discussed below.

Figure 11 shows an instruction cell for non-branching instructions (such as datapath or move instructions). In this cell, the input "active" bit is simply copied to the output. Note that in the instruction cells, both the program location, a, and the instruction word, w, have fixed values.

2.1 Merging Nodes in the Program Netlist

Consider the example of a CFG from Fig. 2. Let us assume that the loop between nodes b and g has a predefined number of iterations. Let us further assume that the branch at node c depends on an external input. Unrolling the model yields a program netlist whose size is exponential in the number of loop iterations. However, if we relocate the jump multiplexer from node c not to the end of the unrolling (as in Fig. 8) but only across nodes d and e to the input of the instruction at node f then the size of the resulting program netlist becomes much smaller (linear in the number of iterations in this example).

Figure 12 shows the program netlist for the unrolling of nodes b to f in Fig. 2.

In general, when we are moving a multiplexer forward through the unrolling we may stop as soon as we reach a specific program location a that is on the execution path of both, the taken and the not-taken branch. This can strongly reduce the number of time frames for which a duplication of logic is needed. We exploit this idea to formulate a combined unroll-and-merge algorithm:

- Unroll the machine program instruction by instruction.
- Create branches at jump instructions and unroll the branches.
- If we encounter a PC value that has been visited before then the corresponding instruction cell is a merge candidate:
  - If merging does not create a cyclic path in the program netlist then do insert a merge cell in front of the already existing cell and merge the branches.
  - If merging would create a cyclic path then do not merge but create a new instance of the instruction cell.

Note that in this algorithm, we are not relocating multiplexers. The program netlist is generated by instantiating instruction cells and merge cells.

2.2 Pruning Dead Branches Automatically

A control flow graph with cycles contains execution paths of infinite length. Accordingly, the program netlist corresponding to such a CFG would be infinitely large. The low-level software that we consider in this work (e.g., tasks with deadlines in a RTOS-based system, interrupt-based drivers, etc.), however, typically has *finite* execution times. In order to obtain program netlists for such software components we have to deal with cyclic paths in the CFG. A typical example is a for loop with a finite number of iterations. Take, again, the CFG of Fig. 2. The branch at node g back to the beginning of the loop is taken $N-1$ times and then, always in the N-th iteration, not taken. We use a SAT-based property checker to identify the branch conditions on the fly when building the program netlist. This is done by automatically generating and checking simple properties on the instances of the *active* signal that is associated with each instance of the program state. Each property checks for a specific active flag instance whether there exists a run of the program for which this flag becomes true, i.e., the corresponding instruction is reachable on the unrolled execution path. In our example, the active flag in the taken branch will be true $N-1$ times and false in the N-th iteration. This information is used to simplify the unrolling. Branches that are determined to be in-active are marked "dead" and will not be unrolled any further.

Note that loop conditions may depend on external inputs. Similarly as in common approaches to hardware verification or worst-case-execution-time (WCET) analysis, user-defined constraints bound the number of unrollings of a loop in such a case.

The resulting program netlist is a compact and accurate representation of all possible execution runs of the program as a combinational circuit. For a given set of assignments to the input variables of the model there is a distinct path of instructions executed – these are marked "active" in the program netlist. The set of all active signals in the program netlist provides global information about the program flow. This is key when applying SAT-based verification to the program netlist. When a SAT solver performs backtracks on the program netlist it can prune out entire execution paths, thus, dramatically improving its run time.

How is the program's computation performed in this representation? It is given by the outputs of the program netlist (a combinational circuit) where the values of registers and program variables depend not only on the input values of the program but also on the set of assignments to the *active* flags representing a particular execution path. This is in contrast to symbolic execution where the computation is represented as symbolic formulas only in terms of program inputs but is generated explicitly for an execution path. There, even at the presence of path pruning and merging techniques, every distinct set of paths needs to be enumerated when solving a verification task. The symbolic expressions for the state variables and the path conditions are generated for every simulation. Each formula is a computational model for a (possibly SAT-based) property checking instance. Also in our approach, all paths are enumerated. However, this is shifted to a pre-processing phase that neglects all of the program's computation that is not relevant to its control flow. No formulas representing the computation along the paths are generated. Afterwards, when solving the actual verification task on the program netlist, the "intelligence" of a SAT solver is used to traverse this implicit representation of execution paths with their associated computation in an efficient way.

A further advantage of this model is that it is compatible with formal hardware verification techniques. Interaction between hardware and software can be easily modeled. Also, when checking properties we can leverage proof technology developed for hardware verification.

3. Modeling Interrupts

In this section, we show how to model interrupt-driven low-level software architectures based on the technique described above. We present the basic idea for a typical interrupt-based software architecture consisting of a set of interrupt service routines (ISRs) and a main program. The ISRs attend asynchronous events produced by external hardware. The main program implements a specific set of tasks serving as the interface to the rest of the software system. It communicates with the ISRs through shared memory variables as in **Fig. 13**. The ISRs also read and write device registers which are modeled as input or output ports.

When building the proposed model the following two steps have to be taken:

(1) Generate the individual program netlists for the program components (ISRs, main program tasks).

(2) Compose and interconnect the instances of the program netlists to create the overall model.

The first step is performed in a straightforward way by calling, for every single CFG belonging to the main program and the ISRs, the program netlist generation procedure explained above. The second step of combining the individual program netlist instances requires additional consideration because, in principle, an interrupt may happen any time (at any instruction). How many times does a particular ISR need to be instantiated in the model and how do we connect the corresponding program netlists to the rest of the model? Clearly, creating an instance of every ISR and inserting it between every two successive instructions that can be interrupted is not feasible.

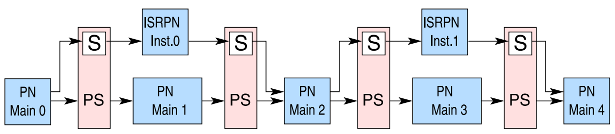

Fig. 13 Example of the model for an interrupt-driven program.

3.1 Simplification by Interrupt Transparency

Let us assume that every ISR correctly implements saving and restoring of the program context. Then, a call of an ISR is *transparent* to the interrupted program, i.e., the architectural state (describing the contents of all relevant CPU registers) for this program is not altered by the interrupt.

We may verify transparency for an interrupt service routine by checking a property directly on the program netlist of the ISR. This is a preprocessing step that needs to be carried out only once for every individual ISR. For a transparent ISR we know that the only effect it can have on the rest of the software system is through the shared memory locations it uses for communication. These locations can be identified easily. Most shared memory variable addresses can be found automatically through simple and local semantic analysis of the program. For more complicated accesses, we can use an all-SAT enumeration to compute all shared memory locations.

We can use these observations in order to simplify our model drastically by instantiating ISRs only once between any two communication operations on the shared memory variables as illustrated in Fig. 13. Let's assume for example that the program netlist (PN) called Main 1 in Fig. 13 contains a sequence of 30 instructions in which the main program does not access the shared memory. The IRQ could happen in between any of the 30 instructions, however, its effect will not be observable by the main program until it reads the shared memory. This happens for the first time in PN Main 2. The actual time point of the IRQ occurring is irrelevant in between these 30 instructions, only that it can occur is important. Therefore, the shared memory variable in the PN is modified between these two well defined states. This is represented as a second combinational logic block (labeled ISR-PN Ins. 0) inserted "in parallel" with the PN of Main 1.

Another benefit of the nature of an interrupt handler is that its program state resides exclusively in data memory so that it can begin execution at an arbitrary architectural state. Therefore, the program netlist representing the ISR is not connected to the architectural state of the interrupted program. This significantly reduces the complexity of the composed model. For example, in Fig. 13, the ISR instances are connected only to the sub-set of the main program's state (PS) that represents the memory (S) shared with the interrupt handler.

3.2 Interrupt Timing

How many instances of an ISR should be inserted between subsequent accesses to shared memory (S) locations? This depends on the HW/SW system being modeled and on the timing of the hardware interrupt events. Remember that we model the software at the ISA abstraction level. Concrete timing information is not included in the model. The interrupt events are modeled through user-defined constraints. In the current implementation of our tool, the user supplies interrupt event constraints by specifying the minimum number of executed instructions between two interrupts. The tool creates as many instances of an ISR between two communication points as are possible according to the constraint.

3.3 Additional Hardware-Dependent Aspects

Modeling interrupts also means modeling priorization and context switching. Interrupt priority resolution schemes are specific to the hardware platform being used. Static and dynamic schemes exist. Context switching is also platform-specific and usually involves saving the return address (current PC), saving the contents of the registers being used by the current program, saving status registers and jumping into the interrupt handler. It depends on the architecture which of these tasks are performed by the CPU hardware and which need to be carried out by the software.

The context switching mechanism implemented in the CPU architecture (such as saving of the return address) is modeled through a platform-specific *IRQ-init cell* that is inserted at the beginning of every program netlist representing an ISR. The IRQ-init cell completes the correct functional behavior of the ISR and is essential for the transparency check.

In our model, priority resolution is represented through a special *IRQ-decision cell*. It is inserted at the program locations determined by the analysis described in Section 3.1 and Section 3.2. Its purpose is to flag its associated ISR netlist instance as active if an interrupt event occurs and priority resolution accepts the interrupt.

4. Experimental Results

The objective of our experimental studies is to show that it is feasible to automatically generate program netlists for realistic hardware-dependent, low-level software such that relevant verification tasks become tractable for formal property checking.

The properties in our experiments describe the functionality of the combined hardware and software, expressed in the behavior of hardware I/O registers and memory cells belonging to the interface of a given program. Propositional logic is used to express conditions on the global I/O signals of the program to be verified.

We developed a software tool, FCK (Formal Checker Kaiserslautern), implementing the program netlist generation technique described above. The front-end of FCK parses the machine code of a given program and stores it as an internal data structure representing the CFG of the program together with data flow information. For the following experiments, the hardware is modeled at the ISA level. The instruction set architecture is provided to the tool in the form of a formalized register transfer description in an input language with a syntax similar to hardware description languages. In its back-end, FCK is interfaced with MiniSat [20] for automatically checking the auxiliary properties for model generation. Besides pruning and unrolling, these engines are also employed to resolve addresses related to indirect memory accesses and to compute target addresses of jump instructions.

For evaluating our approach we formally verified two different programs using FCK. The first program is a software implementation of a synchronous serial receiver (Section 4.1). The second program is an interrupt-based driver implementation of a LIN (Local Interconnect Network) master node (Section 4.3). For both examples, we used a hardware architecture based on the open-source 32-bit RISC processor Aquarius [21]. This architecture implements the SuperH-2 instruction set architecture by Renesas Electronics Corporation.

Table 1 Serial synchronous receiver: Model generation.

	# instructions		program netlist		CPU	Mem.
Program	CFG	PN	# vars.	# clauses	(s)	(MB)
Serial RX	98	6,655	490,030	2,738,762	778	5,194

Section 4.2 describes experiments based on a software for an embedded sensor platform that has been developed to evaluate our fully automatic interrupt model generator.

The program netlists generated by FCK and MiniSat can be used as design-under-verification (DUV) in any hardware verification environment. In the following experiments for proving properties on program netlists we used the commercial tool OneSpin 360MV [22] which is a state-of-the-art hardware property checker. It was used as a black box, no adaptations to our model were made. All experiments were conducted on an Intel Xeon E5440 machine with 16 GB of RAM.

4.1 Synchronous Serial Receiver

The program for the synchronous serial receiver is an example created by ourselves to investigate the behavior of the model generation phase on loop-intensive software. The driver program communicates with a simple I/O device by serially reading values from a 1-bit device data register, performing serial-to-parallel conversion and storing the received data byte-wise in data memory. A new incoming bit is detected after a rising edge on the synchronous input signal. Every bit is sampled three times in order to detect whether it corresponds to a logic one or zero. A message frame contains 32 bits of data. For a finite unrolling of the software we needed to add an assumption based on the maximum idle time on the communication channel as a constraint for model generation, stating that every incoming bit arrives within 5 consecutive read operations.

Table 1 summarizes the results of the program netlist generation phase: The program was written directly in assembly language and contains 98 assembler instructions. After unrolling into a program netlist (PN), the model contains 6,655 instruction cells and 960 inputs (not shown).

The program was verified by means of a safety property covering all possible input scenarios. For each scenario it is checked whether the data has been sampled, converted and stored correctly in memory. This was described by the property as a logical equivalence comparing each bit read from the serial device with the corresponding value stored in data memory. The overall property was then expressed as the logical conjunction of all single comparisons ensuring that all data bits belonging to a serial frame were covered by the property. The check of the property consumed 12 s of CPU time and 10 GB of memory.

4.2 Sensor Platform

A set of drivers for an embedded sensor platform has been created to evaluate our approach to automatically composing pro-

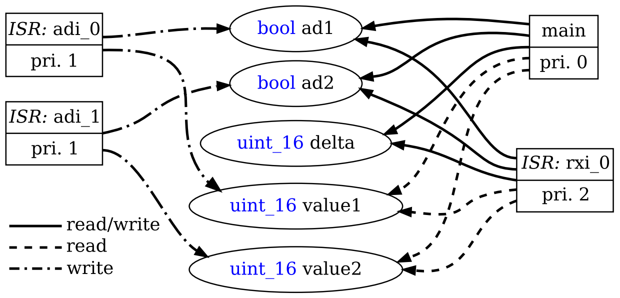

Fig. 14 Sensor platform: Communication structure.

Table 2 Sensor platform: ISR model generation.

program	# instructions		comm.	# instances	
component	CFG	PN	points	optimized	naive
main	100	179	27	1	1
ISR adi_0	12	12	2	21	71
ISR adi_1	12	12	2	21	71
ISR rxi_0	64	68	9	105	1,775

gram netlists of interrupt service routines into a combined model. The main focus of this experiment is to examine the behavior of our model generator for complex communication structures between nested interrupt handlers and a foreground program.

The sensor platform implements a slave connected to a serial communication interface (SCI). It receives two analog values via analog/digital converters (ADC) and performs simple arithmetic operations on these values. The received values and the result are transmitted if a request from a master arrives via SCI. Two ISRs handle the ADCs (adi_0 and adi_1) and the third one takes care of the communication via SCI (rxi_0). The main routine takes the values of the adi-ISRs, performs computations on them and stores the result in the shared memory for the rxi_0. **Figure 14** shows an overview on which software components (boxes in the figure) can access which shared memory locations (bubbles in the figure). The priorities ("pri.") give the information which software component can be interrupted.

Table 2 shows how many PN instances were needed for the composed model and the respective size of those instances. The overall model generation took less than 4 seconds of CPU time. The total size of the assembler program is 161 instructions. There exists only a single instance for the main program. For the ISR an instance is created at every communication point, as explained in Section 3. Due to its priority, rxi_0 can also interrupt the two adi-ISRs. Hence, the more often they are instantiated the more often we need an instance of rxi_0. The result in Table 2 shows that this example has a complex communication structure that may lead to a prohibitively large number of instances for the rxi_0 ISR. However, using the proposed concepts for modeling interrupts only a small fraction of these instances is required (column 4) when compared to the naive approach (column 5) without the simplifications of Section 3.1.

4.3 LIN Master Node Driver

The LIN bus is a communication standard [23] commonly employed in automotive embedded networks. The driver considered in this experiment was developed by Infineon Technologies AG for a node compatible with version 1.3 of the LIN specification.

Table 3 LIN master node driver: Model generation.

program	# instructions		program netlist		CPU	mem.
component	CFG	PN	# vars.	# clauses	(s)	(MB)
LIN-Init	225	385	8,637	45,476	1.32	36
LIN-Main	85	84	1,898	8,760	0.13	27
LIN-ISR	790	1,138	130,499	624,463	11.00	102

We ported the original code of the driver to the Aquarius processor architecture.

In order to provide its functionality, the driver software architecture builds upon an interrupt service routine handling the communication with a UART. Message transfers are initiated by a message scheduler. Every time the transfer of a message field is finished an interrupt request (IRQ) is produced by the UART. Transaction completion is signaled to the message scheduler via shared memory. In total, the LIN driver used for our experiments contains 1,309 lines of C code. The code was compiled using the GNU C compiler.

We created a computational model reflecting the behavior of full bus transactions. In order to build the model we made the following assumptions. The communication between scheduler and the interrupt service routine (ISR) is done only via shared memory, the scheduler is executed once between two consecutive UART interrupts and initialization takes place after system reset.

In the first step, we generated the program netlists for the initialization, the message scheduler and the ISR. **Table 3** shows the results for this step.

To prove the assumption that the ISR communicates only via shared memory the transparency check (Section 3.1) was performed. The transparency property was formally proven. The run time for this proof was less than one second and memory consumption was 132 MB.

Next, we integrated the program netlists into a combined model. The model was created under the constraint that the runtime of the scheduler executed on the CPU is much shorter than the time interval between two interrupt requests, which is a valid assumption for the system under verification. Under this constraint only one instance of the ISR is needed for every run of the scheduler. Subsequent executions of the ISR are represented by concatenations of instances of the ISR program netlist, one for each message field of the LIN frame. The situation is as depicted in Fig. 13, each of the two instances of the ISR in the figure corresponds to the transfer of one message field. The resulting model contains 24,001 instructions and 62,592 primary inputs.

After the model generation phase properties could be checked on the model. The properties we checked on the composed model verify the compliance of the LIN master node with the specification. Run times and memory consumed by the proof engine are shown in **Table 4**.

Each property describes the behavior of the master node for a complete LIN frame. For every field of the LIN frame a logical comparison is performed between the expected protocol values and the corresponding program netlist outputs. Depending on the LIN field, the expected value is a constant or a logical function in terms of the given program netlist inputs. In the case that the node operates in transmission mode (TX) the values written to the

Table 4 LIN master node driver: Property checking.

Property	CPU (s)	Mem. (MB)
RX frame 2 or 4 data bytes incl. checksum	7	1,459
RX frame 4 or 8 data bytes incl. checksum	17	1,641
RX frame 2 or 4 data bytes wrong checksum	7	1,361
RX frame 4 or 8 data bytes wrong checksum	28	1,545
TX frame 2 or 4 data bytes incl. checksum	6	1,206
TX frame 4 or 8 data bytes incl. checksum	15	1,548
Wrong PID or not matching ID	6	1,205
Wrong PID or not matching ID (8 Bytes)	14	1,566

UART transmission buffer are compared with the expected protocol values. Fields such as the message payload and the checksum are expressed as Boolean functions of the input variables corresponding to the data collected from the application interface via shared memory. In receiver mode (RX) output variables corresponding to the data written to the application interface via shared memory are expressed as functions of the UART reception buffer. In this case, input data is represented by means of free variables so that all possible input values are considered in the proofs.

Besides proving the correctness of the generated frames the properties also check the interaction between the software and the UART including correct addressing of peripheral data and control registers as well as their contents.

As can be noted, both, the run times for model generation as well as the proof times for property checking, are quite short.

Note that available tools for software property checking are not applicable to the hardware-dependent verification tasks considered here. For comparison, we also experimented with a standard BMC unrolling that, in principle, could be used to solve the same verification tasks. As already shown in Ref. [16] standard BMC runs out of steam after only a dozen program steps. In fact, running property checking on the program netlist turns out to be more efficient than standard BMC by several orders of magnitude.

5. Conclusion

This paper proposes a *program netlist* as an explicit representation of the program flow and a compact, implicit hardware representation of the program computation for low-level embedded system software. This is achieved by the proposed methodology in which the overall verification task is split into two main phases: program flow analysis and program verification. This supports the compositionality of the proposed model. The program netlist models all possible execution paths (program flow) and represents the program computation by instantiating instruction cells of a given processor architecture.

Furthermore, modeling of interrupt-driven software using program netlists has been introduced in our approach. By using the compositionality of the program netlist it is possible to represent interrupt-driven software. This is achieved by combining individual interrupt service routines (ISR) with the involved foreground program. The composed model is optimized in size by taking into account the transparency of an ISR with respect to the architectural state of the interrupted program. As a consequence, a drastic reduction of the number of needed ISRs instances can be achieved. The transparency criterion can be easily verified using a standard property checker. Our experiments show the feasibility of the proposed methodology.

Future work will explore the application of program netlists in low-level software equivalence checking.

References

[1] Loitz, S., Wedler, M., Stoffel, D., Brehm, C., Kunz, W. and Wehn, N.: Formal Hardware/Software Co-verification of Application Specific Instruction Set Processors, *System Specification and Design Languages*, Kazmierski, T.J.J. and Morawiec, A. (Eds.), Lecture Notes in Electrical Engineering, Vol.106, pp.1–20, Springer New York (2012).

[2] Jhala, R. and Majumdar, R.: Software model checking, *ACM Comput. Surv.*, Vol.41, pp.21:1–21:54 (2009).

[3] D'Silva, V., Kroening, D. and Weissenbacher, G.: A Survey of automated techniques for formal software verification, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* (*TCAD*), Vol.27, No.7, pp.1165–1178 (2008).

[4] Ball, T., Podelski, A. and Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs, *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, London, UK, pp.268–283, Springer-Verlag (2001).

[5] Beyer, D., Henzinger, T.A., Jhala, R. and Majumdar, R.: The Software Model Checker Blast: Applications to Software Engineering, *Int. J. Softw. Tools Technol. Transf.*, Vol.9, pp.505–525 (2007).

[6] Godefroid, P.: Software Model Checking The Verisoft Approach, *Formal Methods in System Design, 2005*, Vol.26, pp.77–101 (2005).

[7] Havelund, K. and Pressburger, T.: Model Checking Java Programs Using Java PathFinder, *Int. J. Softw. Tools Technol. Transf. STTT*, Vol.2, No.4, pp.366–381 (2000).

[8] Holzmann, G.J.: The SPIN Model Checker, *IEEE Trans. Softw. Eng.*, Vol.23, pp.279–295 (1997).

[9] Dill, D.L.: The Murphi Verification System, *Proc. 8th International Conference on Computer Aided Verification* (*CAV '96*), London, UK, pp.390–393, Springer-Verlag (1996).

[10] Schlich, B.: Model checking of software for microcontrollers, *ACM Trans. Embed. Comput. Syst.*, Vol.9, No.4, pp.36:1–36:27 (2010).

[11] Babic, D. and Hu, A.J.: Calysto, Scalable and Precise Extended Static Checking, *Proc. 30th International Conference on Software Engineering* (*ICSE '08*), New York, NY, USA, pp.211–220, ACM (2008).

[12] Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A. and Ashar, P.: Efficient SAT-based bounded model checking for software verification, *Theor. Comput. Sci.*, Vol.404, No.3, pp.256–274 (online), DOI: 10.1016/j.tcs.2008.03.013 (2008).

[13] Clarke, E., Kroening, D. and Yorav, K.: Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking, *Proc. 40th Annual Design Automation Conference* (*DAC '03*), New York, NY, USA, pp.368–371, ACM (2003).

[14] Biere, A., Cimatti, A., Clarke, E.M., Fujita, M. and Zhu, Y.: Symbolic Model Checking Using SAT Procedures Instead of BDDs, *Proc. International Design Automation Conference* (*DAC*), pp.317–320 (1999).

[15] Große, D., Kühne, U. and Drechsler, R.: HW/SW co-verification of embedded systems using bounded model checking, *Proc. 16th ACM Great Lakes Symposium on VLSI* (*GLSVLSI '06*), pp.43–48 (2006).

[16] Nguyen, M.D., Wedler, M., Stoffel, D. and Kunz, W.: Formal Hardware/Software Co-Verification by Interval Property Checking with Abstraction, *Proc. 48th Design Automation Conference* (*DAC '11*), New York, NY, USA, pp.510–515, ACM (online), DOI: 10.1145/2024724.2024843 (2011).

[17] Păsăreanu, C.S. and Visser, W.: A Survey of New Trends in Symbolic Execution for Software Testing and Analysis, *Int. J. Softw. Tools Technol. Transf.*, Vol.11, No.4, pp.339–353 (online), DOI: 10.1007/s10009-009-0118-1 (2009).

[18] Arons, T., Elster, E., Ozer, S., Shalev, J. and Singerman, E.: Efficient Symbolic Simulation of Low Level Software, *Design, Automation and Test in Europe, 2008* (*DATE '08*), pp.825 –830 (online), DOI: 10.1109/DATE.2008.4484776 (2008).

[19] Schlich, B., Noll, T., Brauer, J. and Brutschy, L.: Reduction of interrupt handler executions for model checking embedded software, *Proc. 5th International Haifa Verification Conference on Hardware and Software: Verification and Testing* (*HVC'09*), Berlin, Heidelberg, pp.5–20, Springer-Verlag (2011) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1965974.1965981⟩.

[20] Eén, N. and Sörensson, N.: An Extensible SAT-solver, *SAT*, pp.502–518 (2003).

[21] Aitch, T.: Aquarius: A pipelined RISC CPU (2003).

[22] Onespin Solutions GmbH: Germany, OneSpin 360MV.

[23] LIN Administration: LIN Specification Package Rev. 1.3 (2002).

Bernard Schmidt received his Dipl.-Ing. degree in computer engineering from the University of Kaiserslautern, Germany, in 2007. He is currently working on his Ph.D. degree in electrical and computer engineering in the Electronic Design Automation Group at the University of Kaiserslautern, Germany. His research interest are methodologies and techniques for formal verification of hardware and hardware-dependent software.

Carlos Villarraga received his undergraduate degrees in electrical and electronics engineering in 2002 and a master's degree in microelectronics and computer engineering in 2006 from Universidad de los Andes, Bogota, Colombia. He is currently working toward his Ph.D. degree in electrical and computer engineering in the Electronic Design Automation Group, Department of Electrical and Computer Engineering, the University of Kaiserslautern, Kaiserslautern, Germany. His research interest includes formal verification of hardware-dependent embedded system software.

Thomas Fehmel received his Dipl.-Ing. degree in computer engineering from the University of Kaiserslautern, Germany, in 2012. He is currently working on his doctorate degree in electrical engineering at the same university. His current research field includes hardware-software co-verification with the focus on interrupt-driven embedded systems.

Jörg Bormann holds a Dipl.-Math. degree in mathematics from the University of Karlsruhe and a Ph.D. in electrical engineering from the University of Kaiserslautern. He was with the formal hardware verification groups of Siemens AG and Infineon Technologies. He co-founded OneSpin Solutions for which he developed complete, i.e., gap-free verification, defined the main product and demonstrated quality gain and predictability of this verification technology in large circuit design projects at various companies. With Abstract RT Solutions GmbH he extended this technology to the development of safety critical circuits and to low level software and demonstrated its benefits during the design phase rather than the verification phase. Bormann is now working in system engineering of mobile telecom devices at Intel.

Markus Wedler received his Dipl.-Math. degree in mathematics from the University of Hagen, Hagen, Germany, in 1999 and Ph.D. degree in electrical and computer engineering from the University of Kaiserslautern, Kaiserslautern, Germany, in 2006. From 2006 to 2012 he was a postdoctoral researcher with the Electronic Design Automation Group, Department of Electrical and Computer Engineering, University of Kaiserslautern. He conducted there research in formal hardware verification, particularly formal property verification for datapath and sequential circuits. Since 2012 Markus Wedler is working at Synopsys GmbH, Aachen.

Duc-Minh Nguyen obtained a Ph.D. in electrical engineering from the University of Kaiserslautern in 2009. Afterwards he worked as scientific staff at the University of Kaiserslautern, Germany. He works currently as researcher and lecturer in the School of Electronics and Telecommunications at Hanoi University of Science and Technology. His research activities involve digital hardware design, embedded system design and formal verification of digital designs and embedded systems.

Dominik Stoffel obtained a Dipl.-Ing. degree from the University of Karlsruhe in 1992 and a Ph.D. from the University of Frankfurt in 1999. From 1993 to 1994 he worked as an R&D engineer at Mercedes-Benz in the development of testing methodology for automotive electronics. From 1994 to 1998 he was with the Max-Planck Fault-Tolerant Computing Group in Potsdam. From 1998 to 2001 he was with the Electronic Design Automation group at the University of Frankfurt, Germany. Since 2001 he is working as a research scientist and lecturer in the Electronic Design Automation group at the University of Kaiserslautern. Dominik Stoffel conducts research in the area of design and verification of systems-on-chip. He has a special interest in methodologies and techniques for formal design verification. Dominik Stoffel has received the Award of the German IT Society.

Wolfgang Kunz received his Dipl.-Ing. degree in electrical engineering from the University of Karlsruhe, Germany, in 1989 and Dr.-Ing. degree in electrical engineering from the University of Hannover, Germany, in 1992. From 1993 to 1998, he was with Max Planck Society, Fault-Tolerant Computing Group at the University of Potsdam, Germany. From 1998 to 2001 he was a professor of Computer Science at the University of Frankfurt/Main. In 2001 he joined the Department of Electrical & Computer Engineering at the University of Kaiserslautern. Wolfgang Kunz works in the area of electronic design automation of microelectronic circuits and systems. His current research interests include design methodologies for embedded systems, formal verification of hardware and hardware-dependent software as well as design for power closure. Wolfgang Kunz is a Fellow of IEEE.

(Recommended by Associate Editor: *Hiroki Matsutani*)