GLS-VLSI'99
IEEE Great Lakes Symposium on VLSI
Ann Arbor, Michigan, USA, March 1999
IEEE Computer Society Press, pp. 284-287

A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs

Subhashini Balakrishnan and Sofiène Tahar

Concordia University, ECE Dept., Montreal, Quebec, H3H 1M8 Canada
Email: {subhar, tahar}@ ece.concordia.ca

Abstract

With the increasing emergence of mixed hardware/software systems, it is important to ensure the correctness of such a system formally, particularly for real-time and safety critical applications. We present a hierarchical approach to modeling and formally verifying an embedded system at higher levels of abstraction, using Multiway Decision Graphs (MDGs). We demonstrate our approach on the embedded software for a mouse controller application on a commercial microcontroller (PIC 16C71), using the MDG verification tools. Inconsistencies in the assembly code with respect to the specification, as published in the application notes of the manufacturer, were uncovered through our experiments.

1. Introduction

There are today two main streams of formal verification, theorem proving and model checking [4]. Theorem provers are scalable but are not automatic and require experience and expertise to handle. While a symbolic model checker is automatic, it is restricted to representation at the boolean level and subject to state explosion. *Multiway Decision Graphs* (MDGs) [5] combine the advantages of both of these approaches, and have been successfully applied in the recent past [3].

Embedded systems are finding widespread application, ranging from factory automation to household appliances. In this paper we present a hierarchical approach to modeling and formal verification of a complete embedded system, using *Abstract State Machines* (ASMs) [5], based on MDGs. Our work is motivated by preliminary results obtained from our earlier experimentation [1] with using MDG tools on the microcontroller PIC16C71, commercialized by Microchip Technology Inc., [7]. In [1] we set up formal models for the RT level hardware and the Instruction Set Architecture (ISA) of the PIC16C71, and verify if the instructions in the instruction set are implemented correctly by the microcontroller hardware. We also proposed formal models for the innermost routine (*Bit*) of its embedded software for a mouse controller application [6], and verified the same using the MDG tools. In this paper, we take the next step and present models for the specification and the implementation of the complete mouse controller embedded application software, and demonstrate a hierarchical approach to modeling and formal verification of an embedded system as a whole. We model the system at different levels of the design hierarchy, i.e., the microcontroller RT level, the microcontroller Instruction Set Architecture (ISA), the embedded software assembly code level and the embedded software flowchart specification. We conduct verifications using the MDG equivalence checking procedure between the RTL hardware and the ISA to establish the correctness of the system hardware platform in implementing its intended architecture. We then verify the particular application embedded in the system by checking the equivalence between the assembly code and its intended behavior, specified as a flowchart. Further, we validate our models and verification by conducting property checking.

The rest of the paper is structured as follows: In Section 2, we describe our modeling scheme using MDGs on the microcontroller architecture and its mouse controller embedded software. In Section 3, we summarize the verification of the ISA and demonstrate the verification of the embedded software through equivalence checking and property checking. Section 4 concludes the paper.

2. System Modeling with MDGs

MDGs [5] incorporate variables of *abstract* sort to denote data signals and *uninterpreted function* symbols to denote data operations. MDGs essentially represent relations rather than functions. The MDG system comes with a simple HDL called MDG-HDL used for both structural and behavioral descriptions. MDGs can also represent sets of states, facilitating reachability analysis with abstract representations. Thus, complex embedded systems could be modeled and verified independently of the width of the datapath and that of the registers.

2.1. Microcontroller Architecture

RTL Implementation

The PIC16C71 is an 8-bit microcontroller, including 36 8-bit wide general purpose registers, 15 special function registers and an 8-level deep hardware stack. It has 36 bytes of RAM. An instruction cycle consists of eight Q cycles (Q1 to Q8) [1].

Using MDG-HDL, the 8-bit general purpose registers are modeled using a variable of abstract sort *worda8* instead of a concrete sort with enumeration {0, ..., 255}. The register file operations are described in terms of access functions *read* and *write*, modeled as uninterpreted functions. Similarly, fetching an instruction from the program memory is modeled using the uninterpreted function *fetch*, while ALU functions are expressed using uninterpreted functions e.g., *add*, *sub*, *inc*, *or*. A concrete variable of sort *wordc4*, with enumeration {0, ..., 15} models the Q cycles.

Instruction Set Architecture

The instruction set architecture (ISA) [7] consists of a total of 35 instructions (RISC), each instruction being 14-bit wide. They are categorized as byte-oriented, bit-oriented, literal and control operations. Using MDGs we model the assembly instructions as predicates [1]. The operations (e.g., inclusive-OR) can be modeled using uninterpreted functions *or*, applied to the arguments of the predicates. Predicates can be described using the MDG-HDL basic library function, *transform*. Decoding specific bits of the opcode are modeled using cross functions like *bits_0_to_6* of type [$worda14 \rightarrow wordc7$]. Note that *worda14* is a 14-bit abstract sort while *wordc7* is a 7-bit concrete sort.

2.2. Embedded Software

The implementation of a serial mouse using the PIC16C71 [6] is taken as our case study. The major tasks performed by the mouse controller software are Button scanning, X and Y motion scanning, and formatting and sending data to the host. The software is composed of three routines: *Main, Byte*, and *Bit*. The *Main* routine (Figure 1) detects any changes in the button status and in the movement of the mouse. It calls the routine *Byte* five times to format and send five bytes of data to the host. The *Byte*, in turn, calls the routine *Bit* eight times to get 8 bits for each byte of data. The routine *Bit* consists of two subroutines, *Bitx* and *Bity*. The *Bit* counts the number of pulses from the outputs of the photodetectors and determines the direction (X or Y) of movement.

Flowchart Specification

We model the flowchart specification of the embedded software as an Abstract State Machine (ASM), each state of which characterizes the contents of the microcontroller registers. The instructions in an assembly code or the statements of a specification control the transitions from one state to another. The registers, e.g., *XCount* and *YCount* are modeled using the abstract sort *worda8*. An abstract function *bitset* of type [$worda8 \times word\mathrm{c}3 \rightarrow worda8$], where *word*c3 is a concrete sort with enumeration {0, ..., 7}, is used to set a particular register bit. A cross function *isbitzero* of type [$worda8 \times word\mathrm{c}3 \rightarrow bool$] is used to test if a particular bit (specified by a 3-bit address) of a register of abstract sort *worda8* is reset or set.

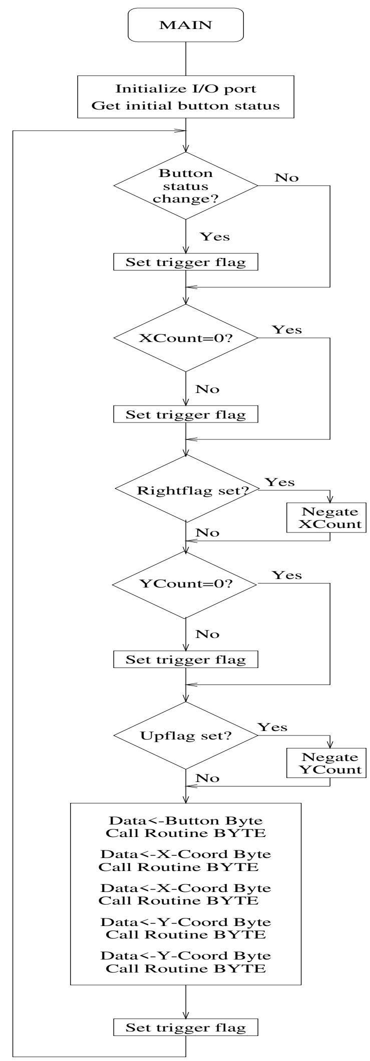

Figure 1. Flowchart specification of *Main*

Assembly Code Implementation

The assembly code implementation is modeled as a set of instructions implementing the control behavior of a routine, which is again represented as an ASM. Figure 2 shows an excerpt of the assembly language code of the routine *Bitx*.

Instruction variables such as *RA.b2* and *CSTAT.b2* are modeled using concrete sort *bool*, and *pc* using sort *wordc5*.

Furthermore, we use the same abstract and cross functions as in the flowchart specification. This is mandatory when checking equivalence using the MDG system.

```
....
BTFSS RA.b2
GOTO BIT0
....
BTFSS RA.b3
....
BSF FLAGB.b3
GOTO BITY
BTFSS CSTAT.b2
GOTO BITY
....
BSF FLAGB.b3
```

Notations:

```
BTFSS Ri.bj: test bit j of Ri. skip next inst. if set
GOTO L: jump to the address indicated by label L
BSF Ri.bj: set bit j of register Ri
```

Figure 2. Excerpt of the assembly code of Bitx

3. Hierarchical Verification in MDG

The MDG software tools package algorithms for computing disjunction, relational product, pruning-by-subsumption, and reachability analysis [5]. Based on these algorithms, the MDG tools provide procedures for combinational verification, sequential verification, safety property checking, and counterexample generation. The sequential verification provides invariant checking and equivalence checking of two state machines. We employ the latter three procedures for our verification experiments in this work.

We adopt a bottom-up approach by first verifying the correctness of the microcontroller hardware (RT level) in implementing its instruction set architecture. Having ensured the fact that each of the instructions in the instruction set is correctly executed in the hardware, we then proceed to verify the embedded assembly program against its behavioral specification. We thus pave a way for automatic verification of an embedded system hierarchically at higher levels of abstraction. All our experiments were conducted on a Sun Ultra Sparc 1 with 252 MB of main memory.

3.1. Formal Verification of the ISA

The instruction set architecture of a microcontroller is the specification of the effect that each instruction is intended to have on the visible state which consists of the visible registers and memory. To verify a microcontroller implementation against its instruction set is to verify that the hardware execution of every instruction has the intended effect.

In PIC 16C71 the instruction execution is signalled by the starting of the Q1 cycle. To verify the microcontroller architecture, we check the equivalence between the circuit machine (ASM) consisting of the microcontroller RTL implementation with that of the ISA, where each transition of the RTL implementation corresponds to the execution of an instruction as specified by the architecture [1]. With the MDG tools this is achieved through sequential equivalence checking, which ensures if the outputs of the two machines are the same at every clock cycle. Applying our models, we verified the following nine instructions, spanning all the four categories of operations in the ISA: INCF, DECFSZ, BSF, BCF, BTFSS, BTFSC, GOTO, CALL and RETLW. The results of the verification including the total CPU time, memory usage and number of MDG nodes generated are given in Table 1.

Table 1. Equivalence checking on system h/w

Verification	CPU time (sec.)	Memory (MB)	MDG Nodes allocated
RTL and ISA	14.27	9.15	21940

3.2. Formal Verification of the Embedded Software

Equivalence Checking

Here again, we follow a hierarchical approach for our verification. We start by verifying the subroutines *Bitx* and *Bity* of the routine *Bit*. We then abstract away these verified subroutines by replacing them by their respective specifications. This approach effectively reduces the state space of the product machine to be verified and hence the verification CPU time and memory usage (see Table 3). We subsequently verify the routines *Bit*, *Byte* and *Main* hierarchically. We divided the *Main* routine into three segments (Figure 1). The first segment consists of all operations before passing data from *Main* to *Byte*. The second segment consists of passing data to *Byte* and making the first call to this latter. Resetting the trigger flag and the backward looping constitutes the third segment. *S*ubsequent calls to *Byte* are similar to the second segment. It therefore suffices to verify one routine call, thus removing any redundant verification.

The equivalence checking between the behavioral specification and the embedded software showed an error in the assembly language code, indicated by a counterexample generated by the MDG tools, providing a trace leading to the software error. Accordingly, one of the instructions, BTFSS (highlighted in Figure 2) is to be replaced by BTFSC. This result confirms with the one obtained in [8] using the SMV. Table 2 summarizes the performance statistics for the equivalence verifications on the original embedded software, up to the *Bit* routine.

Table 2. Eq. checking on original embedded s/w

Routine	Counterex. gen. (sec.)	CPU time (sec.)	Memory (MB)	MDG Nodes allocated
BIT1 of BITX	0.43	0.18	1.3	918
BITX	0.12	0.06	9.23	491
BIT	0.22	0.62	1.89	2864

The erroneous instruction is corrected and the equivalence checking is run successfully. Table 3 shows the performance statistics for the hierarchical equivalence verifications of all routines.

Table 3. Eq. checking on corrected embedded s/w

Routine	CPU time (sec.)	Memory (MB)	MDG Nodes allocated
BIT0 of BITX	0.12	1.33	783
BIT1 of BITX	0.11	2.05	783
BITX	0.10	7.57	357
BIT0 of BITY	0.21	1.33	783
BIT1 of BITY	0.17	1.66	783
BITY	0.07	7.57	357
BIT	0.61	2.73	3412
BYTE	22.86	1.00	26485
MAIN	25.67	6.11	38167

Property Checking

The MDG tools allow safety property checking as invariants. Therefore, apart from conducting equivalence checking, we also validated our models and verification results by checking key properties on the registers. Following are two example properties that test the status of the *Right Flag* in accordance with the primary inputs:

Property 1: **If** RA.b2 ∧ [¬ CSTAT.b2]
then [¬ Rightflag]

Property 2: **If** RA.b2 ∧ [¬ CSTAT.b2] ∧ [¬ XDATA]
then Rightflag

where *RA.b2* and *CSTAT.b2* denote bit 2 of the registers *RA* and *CSTAT*, respectively, and *XDATA* is the output from one of the phototransistors.

In MDG-HDL, we first transform the *if* and *then* statements, each into a *Directed Formula* (DF) [5]. A DF can be represented in terms of a circuit. Thus, a property is represented by two equivalent circuit forms, each of which is described as an ASM in MDG-HDL. The two circuits corresponding to a property are composed and run concurrently with the state machine on which the property is to be checked. The output of the circuit corresponding to the condition statement is compared with the output of the circuit corresponding to the *then* statements. The equivalence between the two is checked as an invariant.

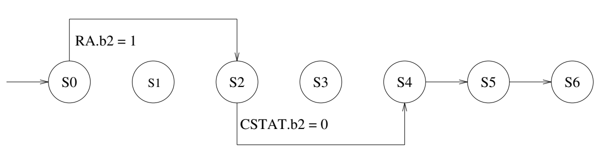

Figure 3. State machine for RA.b2=1 & CSTAT.b2=0

We verified the above properties 1 and 2 on the routine *Bit*. A segment of state machine of *Bit* corresponding to the inputs in property 1 is traced in Figure 3. The property verification discovered the error matching with that shown by the equivalence checking. Performance statistics for the property verification on both the original and corrected implementation of the routine *Bit* is shown in Table 4.

Table 4. Property checking on *Bit* routine

Property	Counterex. gen. (sec.)	CPU time (sec.)	Memory (MB)	MDG Nodes allocated
P#1 on org.	-	1.37	1.08	592
P#1 on cor.	-	1.34	1.06	567
P#2 on org.	0.10	1.44	1.10	711
P#2 on cor.	-	1.37	1.08	613

4. Conclusions

In our earlier work [1], we experimented with using MDGs on the hardware of a commercial microcontroller, PIC16C71 and a small section (routine *Bit*) of its embedded software application for a mouse controller application. In this paper, we succeeded in verifying the entire software on the microcontroller. We thus demonstrated a hierarchical approach to modeling and verification of a commercial microcontroller ISA and its embedded mouse controller software, using MDGs. As shown in the verification results, the experiments concluded in few seconds of CPU time. Our approach hence effectively enables an entire embedded system to be modeled and verified at different hierarchical levels automatically using the MDG tools.

References

[1] S. Balakrishnan and S. Tahar. “Modeling and Formal Verification of a Commercial Microcontroller for Embedded System Applications”. *Proc. IEEE 10th International Conference on Microelectronics* (ICM’98), Monastir, Tunisia, December 1998, pp. 107-110.

[2] K. Buchenrieder, A. Sedlmeier, and C. Vieth. “HW/SW Co-Design with PRAM’s using CODES”. *Proc. Computer Hardware Description Languages and their Applications* (CHDL’93). Elsevier Science Publishers B. V., 1993, pp. 65-78.

[3] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. “Automated Verification with Abstract State Machines Using Multiway Decision Graphs”. In: Kropf, T. (Ed.), *Formal Hardware Verification: Methods and Systems in Comparison*, LNCS 1287, Springer Verlag, 1997, pp. 79-113.

[4] E. Clarke and J. Wing, Formal Methods. “State of the Art and Future Directions”. *CMU Computer Science Technical Report* CMU-CS-96-178, August 1996.

[5] F. Corella, Z. Zhou, X. Song, M Langevin, and E. Cerny. “Multiway Decision Graphs for Automated Hardware Verification”. *Formal Methods in System Design*, Vol. 10, No. 1, 1997, pp. 7-46.

[6] Microchip Technology Inc. “Embedded Control Handbook”, 1993, pp. 2.121-2.133.

[7] Microchip Technology Inc. “PIC16C71”, 1994, pp. 2.328-2.372.

[8] O. Thiry and L. Claesen. “A Formal Verification Technique for Embedded Software”. *Proc. IEEE International Conference on Computer Design* (ICCD’96), Austin, Texas, USA, October 1996, pp. 352-357.