Model checking of embedded assembly program based on simulation

PAPER

Model Checking of Embedded Assembly Program Based on Simulation

Satoshi YAMANE[†a)], ***Member***, **Ryosuke KONOSHITA**[†], ***and*** **Tomonori KATO**[†], ***Nonmembers***

SUMMARY Embedded systems have been widely used. In addition, embedded systems have been gradually complicated. It is important to ensure the safety for embedded software by software model checking. We have developed a verification system for verifying embedded assembly programs. It generates exact Kripke structure by exhaustively and dynamically simulating assembly programs, and simultaneously verify it by model checking. In addition, we have introduced undefined values to reduce the number of states in order to avoid the state space explosion.

key words: *embedded assembly program, model checking, simulation*

1. Introduction

Embedded systems are widely used in airplanes, cars, and household appliances. It is important to find errors and repair them. Model checking [1] is useful for this purpose. Recently software model checking [2] is actively studied, and program verification [3] is receiving a lot of attention. B.Schlich have developed model checking [MC]SQUARE [10], [11] of assembly programs for microcontrollers. [MC]SQUARE generates overapproximated models by static program analysis, and verifies them by model checking. This model checking can verify assembly programs, and find various errors such as stack overflow and stack underflow.

In this paper, we develop new model checking of assembly programs. While we generate an exact model by dynamic program analysis, simultaneously verify it. The reasons to verify assembly programs are as follows:

1. We realize program verification at the level of registers. We can verify hardware-dependent erros such as for example, stack overflows, stack underflows and interrupt handing erros.

2. We realize verifying timing errors. We can estimate the execution time of assembly programs.

But verifying assembly programs causes the state space explosion problem [4]. B.Schlich generates the whole overapproximated models by static program analysis, and after that verifies them by model checking [MC]SQUARE. But B.Schlich does not consider clock cycles.

In this paper, we generate Kripke structure such as the exact models including clock cycles, and develop abstract and refinement method of the bit level by undefined values. Also we verify Kripke structure by model checking while generating the Kripke structure by dynamic program analysis.

We explain our proposed new methods as follows:

1. By generating the exact models including clock cycles, we can uniquely decide the timing of the interrupt about clock cycles. Therefore we can reduce the number of states of Kripke structure. Moreover we can verify timing constraints.

2. Our proposed abstract and refinement method of the bit level is quiet different from Delayed NonDeterminism (DND) [12]. In our method, only bits needing concretization is refined. Therefore we avoid the state space explosion problem.

3. By the exact Kripke structure, we never judge it to be dangerous when it is safe.

4. As we verify Kripke structure by model checking while generating the Kripke structure by dynamic program analysis, verification results may be provided even if we do not generate the whole Kripke structure. Therefore we may avoid the state space explosion problem.

We demonstrate the effectiveness of our proposed verification method for robots [6] which carried microcomputer H8/3687[5] of Renesas company. In addition, this robot is equipped with plural timers and analog-digital converters.

The rest of this paper is structured as follows. First, Sect. 2 introduces Kripke structure and model checking. Our proposed verification system is described in Sect. 3. Especially, we describe interrupts and an abstraction method. Experiments of embedded robot software are described in Sect. 4. Finally, Sect. 5 concludes this paper.

1.1 Related Works

B.Schlich reported that embedded C programs were not verified by the existing C code model checkers (e.g. BLAST [7], BOOP [8]) [9] because embedded C contains more features than defined in ANSI C.

Afterwards B.Schlich developed model checker [MC]SQUARE, which verified assembly programs [10]. [MC]SQUARE generates the whole overapproximated model by static program analysis, and then verifies it by model

Manuscript received November 7, 2016.
Manuscript revised March 2, 2017.
Manuscript publicized May 12, 2017.

[†]The authors are with Graduate School of Natural Science and Technology, Kanazawa University, Kanazawa-shi, 920–1192 Japan.
a) E-mail: syamane@is.t.kanazawa-u.ac.jp

DOI: 10.1587/transinf.2016EDP7452

checking. But [MC]SQUARE does not consider clock cycles. B.Schlich developed abstraction techniques such as Delayed NonDeterminism (DND) [12], Dead Variable Reduction (DVR) [13], [14], Path Reduction (PR) [14] in [MC]SQUARE. DND is an abstraction technique that is used when replacing abstract values with concrete values.

In this paper, our proposed method is quiet different from [MC]SQUARE as follows: (1) Generating models including clock cycles, (2) Abstract and refinement method of the bit level, (3) Generating exact models by dynamic program analysis, (4) Verifying a model by model checking while generating the model by dynamic program analysis.

On the other hand, Lynette Millett sliced the Promela programming language, used to specify protocols for the Spin model checker [15]. A static program slice consists of the parts of a program that may affect or are affected by the value being computed at the point of interest. Our method is dynamic abstract and refinement method of the bit level, which is quiet different from Lynette Millett's method.

2. Overview of Kripke Structure and Model Checking

We define Kripke structure [16] as the model generated from assembly program, and describe model checking [1].

Let AP be a set of atomic propositions. A Kripke structure M over AP is a three tuple $M = (S, R, L)$ where

- S is a finite set of states.
- $R \subseteq S \times S$ is a transition.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

We use CTL(Computational Tree Logic) for specifying properties of Kripke structures [17]. CTL formulas are composed of path quantifiers and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers A("for all computation paths") and E("for some computation path"). On the other hand, the temporal operators describe properties of a path through the tree. There are five basic operators such as X("next time"), F("eventually" or "in the future"), G("always" or "globally"), U("until") and R("release").

Given a Kripke structure $M = (S, R, L)$ and a temporal logic formula ϕ, find the set of all states in S that satisfy ϕ.

In this paper, we verify whether stack overflow happens or not. We specify stack overflow by CTL [17] as follows.

$$AG(s_{STACK} \leq LIMIT_{STACK}) \quad (1)$$

$$= \neg EF(s_{STACK} > LIMIT_{STACK}) \quad (2)$$

,where s_{STACK} denotes the consumption of the stack in some state, and $LIMIT_{STACK}$ denotes the use limit quantity of the stack. This formula intuitively means that $s_{STACK} \leq LIMIT_{STACK}$ holds at every state on every path from initial states; that is, $s_{STACK} \leq LIMIT_{STACK}$ holds globally.

In this paper, we verify $EF(s_{STACK} > LIMIT_{STACK})$. That is, if $EF(s_{STACK} > LIMIT_{STACK})$ does not hold true at initial states, $\neg EF$ holds true. In this case, stack overflow does not happen.

We can easily verify other properties described in CTL.

3. Verification System

3.1 Overview of Verification System

This subsection describes the configuration of the verification system, which consists of Simulator and Model Checker as shown in Fig. 1.

First Simulator inputs assembly program, and generates a Kripke structure. Next Model Checker inputs the Kripke structure and a property, and outputs true or false. Especially, Model Checker inputs a Kripke structure while Simulator generates the Kripke structure.

Simulator generates the exact model of the behavior exhibited by the correponding assembly program, based on dynamic program analysis. The exact model is described by Kripke structure, which consists of a finite set S of states, a transition $R \subseteq S \times S$ and a set of atomic propositions. The set of atomic propositions denote input and output informations from environments, events, registers. For example, n-th register is described by $Reg(n) = XXXX$, a memory value by $add = XXXX$, a stack pointer by $stack = XXXX$, a program counter by $PC = XXX$. In addition, PC in a state s is denoted by $s.PC$.

3.2 Algorithm of Verification System

The algorithm of our verification system is defined by Algorithm 1.

First we explain the outline of Algorithm 1.

1. First, in an initial state s_0, all enabled interruptions are executed by INTERRUPTHANDLING, and then INTERRUPTHANDLING generates scucesosr states (line 10,23). A generated state s' by INTERRUPTHANDLING(line 10) is added to Kripke structure by ADDNEWSTATE (line 31,43). Afterwards MODELCHECKEF verifies the Kripke structre by model checking (line 47,50). We assume an interrupt processing is one instruction.

2. Next, after interruptions, the instruction of the address of program counter PC in a state s is executed, and then the next state s' is generated (line 12,37). A generated state s' by INTERRUPTHANDLING(line 10) is added to Kripke structure by ADDNEWSTATE (line 40,43). Afterwards MODELCHECKEF verifies the Kripke structure

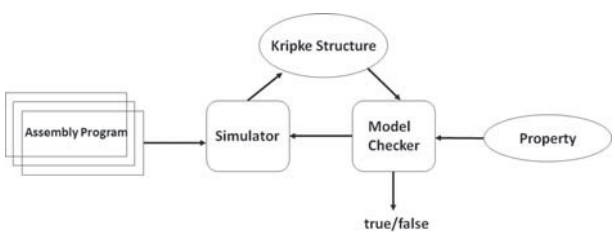

Fig. 1 Configuration of verification system

Algorithm 1 Algorithm of verification system

```
1: f := s.STACK > LIMIT_STACK                          ▷ Formula
2: EFf                                                 ▷ Property
3: s_0                                                 ▷ initial state
4: S := {s_0}                                          ▷ set of states
5: R := ∅                                              ▷ set of relations between states
6: list = [s_0]                                        ▷ generated states
7: function MAIN
8:     while list.length ≠ 0 do
9:         s ← head of list                            ▷ current state s
10:        INTERRUPTHANDLING(s)                        ▷ generate state by interrupt
11:        if decidable interrupts don't exist then
12:            EXECUTEINSTRUCTION(s)                   ▷ generate state
13:        end if
14:        if EFf ∈ L(s_0) then break                  ▷ verification terminates
15:        end if
16:        remove s from list
17:    end while
18:    if EFf ∈ L(s_0) then return (S, R, true)        ▷ output stack overflow
19:    else return (S, R, false)                       ▷ don't stack overflow
20:    end if
21: end function
22:
23: function INTERRUPTHANDLING(s)
24:    for all i ∈ Interrupts do
25:        if i is interruptible then
26:            s' ← s                                  ▷ Generate new state s' moved from s
27:            PC_i = VectorTable[i]                   ▷ get vector address
28:            s'.PC = PC_i                            ▷ set PC_i to PC of s'
29:            GlobalMaskBit_s' ← true                 ▷ mask s'
30:            InterruptFlag_s' ← false                ▷ clear flag of s'
31:            ADDNEWSTATE (s, s')
32:            EXECUTEINSTRUCTION (s')                 ▷ interrupt is executed
33:        end if
34:    end for
35: end function
36:
37: function EXECUTEINSTRUCTION(s)
38:    operation ← memory[s.PC]                        ▷ get operation accroding to PC
39:    s' ← execute(s, operation)                      ▷ generate a new state
40:    ADDNEWSTATE (s, s')
41: end function
42:
```

```
43: function ADDNEWSTATE(s, s')
44:    S := S ∪ {s'}                                   ▷ add new state to S
45:    R := R ∪ {(s, s')}                              ▷ add new transition from s to s'
46:    add s' at the tail of list
47:    MODELCHECKEF (s')
48: end function
49:
50: function MODELCHECKEF(s)
51:    T := φ
52:    if s.STACK > LIMIT_STACK then                   ▷ formula f holds ture
53:        T := T ∪ {s}
54:    end if
55:    while T ≠ φ do
56:        Choose {s ∈ T}
57:        T := T/{s}
58:        L(s) := L(s) ∪ {EFf}
59:        for all t such that R(t, s) do
60:            L(t) := L(t) ∪ {EFf}
61:            T := T ∪ {t}
62:        end for
63:    end while
64: end function
```

by model checking (line 47,50).

While a new state is generated, that is, while *list* is not empty, Algorithm 1 repeats the above procedure. But when $s_0 \in L(EFf)$ holds true, MODELCHECKEF outputs true, and then terminates.

Next we explain main functions in Algorithm 1.

1. In INTERRUPTHANDLING(line 23), interruptions are executed. The top address of the interrupt service routine corresponding to an enabled interrupt *i* is captured from the interrupt vector table, and then is substituted for *PC* (line 27). Afterwards flags are masked (line 29) and released (line 30), and then the interrupution is executed.

2. In EXECUTEINSTRUCTION (line 37), a new next state is generated. In EXECUTEINSTRUCTION (line 37), there are two functions as follows.

   a. In $execute(s, operation)$ (line 39), a new next state s' is generated by updating propositions in current state$s$ corresponding to an input instruction $operation$. For example, we explain move instruction between registers and registers. (1) First a source register is refined in order to concretize values of CCR, (2) Next the value of the source register is moved to the value of a destination register, and then CCR is set, (3) Finally both a timer counter and PC are updated.

   b. In ADDNEWSTATE (line 43), a new generated state s' is added in Kripke structure. (1) First s' is added in the set of states, and the transition relation between s and s' is added in the set of relations (line 44,45). (2) Next s' is added in *list* (line 46). (3) Finally new updated Kripke structure is verified by model checking (line 47).

3. Whenever *Simulator* generates a new state, MODELCHECKEF (line 50) is performed. (1) First MODELCHECKEF (line 50) checks whether the stack pointer in a state s exceeds the stack domain (line 52). If the stack pointer does not exceed the stack domain, nothing is dones. Otherwise, s is added into a set T (line 53), (2) Next until T is empty (line 55), a state s is chosen from T (line 56), and s is deleted from T (line 57), (3) For any state t which satisfies $R(t, s)$ (line 59), EFf is added in $L(t)$ (line 60) and t is added in T (line 61).

Example 1: If $s_0 \in L(EFf)$, stack overflow is detected (line 18).

For example, we explain simulation and model checking by Fig. 2.

First Simulator executes MOV.W, and generates a new state s'. Next whether s' satisfies f or not is checked. When we suppose that s' does not satisfy f, Simulator executes PUSH.W, and generates a new state s''. When we suppose that s'' satisfies f, EFf is added in $L(s')$ which satisfies $R(s', s'')$. Moreover EFf is added in $L(s)$ which satisfies

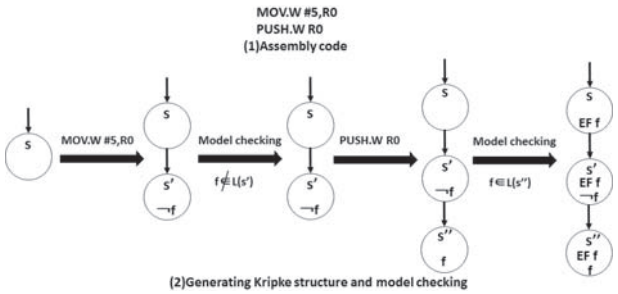

Fig. 2 Example of simulation and model checking

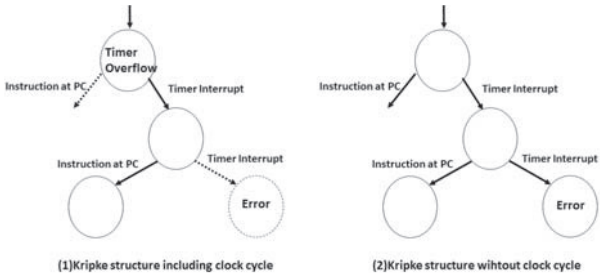

Fig. 3 Kripke structure either including or without clock cycles

$R(s, s')$.

3.2.1 Interrupts

All interrupts have fixed priorities. The interrupt stops a current processing temporarily and performs a high processing of importance earlier. At 25 line in Algorithm 1, whether the interrupt is enabled or not is checked.

We explain how to check whether the interrupt is enabled or not as follows.

When many interrupts are enabled, an enabled interrupt is chosen from four elements such as (A)whether there is an interrupt request, (B)the interrupt enable bit, (C)whether the timing of the interrupt is decided, (D)the priority of the interrupt.

In the following, we consider the timer interrupt that assumed the overflow of the timer trigger .

1. When a timer overflows, this timer interrupt starts the demand of the interrupt by doing an appointed bit truly. Without this demand, the interrupt is not executed (A).
2. Even if a demand of the handling of an interrupt is given, the interrupt is not executed if it is not allowed by Interrupt Enable bit (IE) and Global Mask bit (GM) (B).
3. By the interrupt that can decide a timing (C), interrupts having priorities which are lower than the interrupt are not executed (D).

The timing of the interrupt can be decided as follows. We consider the timer interrupt that assumed the overflow of the timer trigger as an example.

$$if_s = (tc_s + c_i > lim_s) \vee if_{s'} \tag{3}$$

$$ti = if_s \wedge \neg gm_s \wedge ie_s \tag{4}$$

1. We check a timer overflow by Eq. (3). if_s denotes Interrupt Flag(IF) at a current state s. As a result of having calculated the right side of Eq. (3), IF becomes true if overflow occurs, where tc_s is a timer count at a current state s, c_i is the number of clock cycles necessary to execute current instruction i, lim_s is the ceiling value for a timer at a current state s, $if_{s'}$ is Interrupt Flag(IF) at a previous state s'. If if_s is true, there is an interrupt request.
2. Equation (4) denotes an enabled condition of the interrupt, where gm_s is Global Mask bit (GM), ie_s is Interrupt Enable bit (IE). If ti is true, the interrupt is executed.

From Eq. (3) and Eq. (4), we can decide the timing of the interrupt. If we can decide timings of all interrupts, we can generate Kripke structure, which is an exact model.

Example 2: For example, we consider two Kripke structures such as Fig. 3(1) and Fig. 3(2) generated from the same assembly program. Kripke structure including clock cycles is shown in Fig. 3(1), and Kripke structure without clock cycles is shown in Fig. 3(2). Here we consider only timer interrupts. By generating Kripke structure including clock cycles, we can decide timings of timer interrupts by a timer count, Global Mask bit (GM) and Interrupt Enable bit (IE).

1. In Fig. 3(1), a timer overflow happens in a top node. In this case, states and transitions by executing the timer Interrupt are generated. Next states and transitions by executing usual instructions of program counter(PC) are generated.
2. On the other hand, in Fig. 3(2), both a timer interrupt and usual instructions at all the nodes are generated. In this case, the Kripke structre reachs an error state. But this transition to the error state may be spurious.

It is important to consider clock cycles in order to both generate an exact Kripke structure and avoid spurious counterexamples.

3.2.2 Undefined Values

Abstraction and refinement method by undefined values is based on DND [12]. An undefined value X denotes 0 or 1. Because an undefined value can make two states one when we pay my attention to one bit, undefined values can contribute to reduction of the number of states.

For example, undefined values are used as shown in Fig. 4. R0 shows that lower 8 bits are undefined in a 16-bit register. @FF21 shows that all bits of address H'FF21 on the memory are unclear. An undefined bit is described by

R0 00000000XXXXXXXX
@FF21 XXXXXXXX

Fig. 4 Undefined values

an undefined value. This undefined bit describes the initial value given from environments. In addition, we treat the undefined values with being unsettled as possible. When an undefined value is divided into 0 and 1 when an instruction accesses it. The refinement is performed with the execution of an instruction in 39 line of Algorithm 1 mainly.

The exhaustive search of Simulator is performed while an initial value is an undefined value and it is gradually refined. Particularly, such an undefined value becomes essential because it is unknown as for the value at the time of the power supply injection as for embedded systems unless we state an initial value clearly.

Example 3: We explain processes of refinement using Fig. 5.

Figure 5 shows how to handle undefined values when we transfer 8 bits data from register R1 to register R0. When an instruction is executed because all bits are undefined values, the necessary part of R1 must be refined. The part which is necessary here is Condition Cord Register(CCR) affected by data transmission. Thus, by the relation between CCR and transfer data, we refine it not to cause contradiction.

The result of the instruction is stored in CCR. CCR in H8/3687 is comprised of 8 bits, and each bit changes depending on an execution. By data transmission, a negative flag (N flag) and a zero flag (Z flag) change. We can judge the setting of the N flag if we watch Most Significant Bit (MSB). If any bit contains 1, we understand that Z flag is not 0.

First we break off the contradiction for the Z flag. In this example, we can divide all the cases into nine cases when R1 includes 1 in either bit and when all bits of R1 are 0. Next we refine N flag. In this example, N flag is true when MSB of R1 is 1, N flag is false when MSB of R1 is 0, N flag is an undefined value when MSB of R1 is an undefined value. Because there is not contradiction in nine cases, the refinement of the N flag is unnecessary.

After that, because all the contradictions are removed, we transfer data from R1 to R0.

Without undefined values, in Fig. 5, we must generate 2^8 transitions (because the transfer of 8 bits data). With undefined values, we can generate nine transitions. Therefore an undefined value is effective for reducing the number of states and transitions. On the other hand, there may not be approximately an effect by some instruction. For example, in the case of add instruction and sub instruction, it becomes difficult to break off contradiction with CCR by partial refinement.

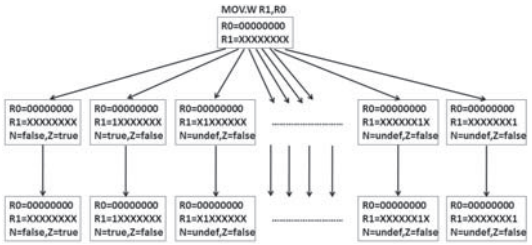

Fig. 5 The procedure of refinement

4. Experiments of Verification System

4.1 Embedded Software

The experiment of our verification system demonstrates the effects of our proposed techniques. We used seven programs written for H8/3687 microcontroller [5], [6].

Program 1: LED program lights up three LEDs by the number of timer overflow interrupts of timer V. For example, when five times of timer overflow interrupts occur, the program lights up LED1 and LED3.

Program 2: PID program operates a motor until it arrives at the aim. The motor is controlled by PID control, and the targeted value is decided beforehand.

Program 3: stack program calculates the numerical sum to 1-255 by recursive function. When Simulator detects stack overflow, it terminates and outputs Kripke structure.

Program 4: Tsensor_LED program acquires a combination of outputs of sensors, and lights up LEDs. This processing to let supporting LED turn on is described as a timer overflow interrupt of timer B1. Here there are three LEDs and three sensors, and the sensor can distinguish black and white. For example, when sensor 1 and sensor 2 detect black, program lets LED1 and LED2 turn on in Tsensor_LED program.

In addition, the LED turns off the light every uniformity time. This processing is described as a timer overflow interrupt of timer V.

Program 5: Tsensor_motor program acquires the value of the sensor, and lets a motor work based on the value. When a timer overflow interrupt occurs, the program acquires the value of the sensor. After acquiring the value of the sensor, the program decides a current value to cancel in a motor and hands the value to the motor. There are three sensors and motors. A sensor is a thing same as sprogram 1, and a motor is a thing same as program 2.

Program 6: Tsensor_P program acquires the value of the sensor only once and decides the targeted value. Afterwards, the rule number of timer interrupts happens, and the software moves a robot. Here there are three sensors and motors. A sensor is a thing same as program 1, and a motor is a thing same as sprogram 3.

Table 1 Embedded software

Program	C code (lines)	Assembly Code (lines)
LED	32	107
PID	141	510
Stack	8	42
Tsensor_LED	42	118
Tsensor_motor	34	100
Tsensor_P	90	272
Linetrace	249	811

Program 7: Linetrace program acquires the value of the sensor, and the program operates H8/3687 microcontroller [5] from the value. Here there are three sensors and a motor. A sensor is a thing same as program 1, and a motor is a thing same as program 3. When a timer overflow interrupt of timer B1 occurs, program acquires the value of the sensor, and sets the new current targeted value from the value. When a timer overflow interrupt of timer V occurs, program performs PID control from the current targeted value and the current value, and outputs the value in a motor.

We show the number of lines of seven above-mentioned C language program and the assembly program in Table 1.

4.2 Results of Experiments

4.2.1 Overview of Experiments

Our proposed verification system has the following originality: (1) generating models including clock cycles, (2) abstract and refinement method of the bit level, (3) generating exact models, (4) verifying a model by model checking while generating it by dynamic program analysis. We show them effective by experiments as follows:

1. We compare (4)"verifying a model by model checking while generating the model by dynamic program analysis" with "verifying a model after generating the model", using only stack program. When we verify a model by model checking while generating the model by dynamic program analysis, we show how much the number of the states can reduce by changing program stack size.

2. We implement both verification systems when we do not consider a clock cycle and when we consider a clock cycle, and compare the difference with both.

3. We compare the difference with three cases as follows. (1) When we use undefined values for all, we generate Kripke structure. (2) When we do not use undefined values for all, we generate Kripke structure. (3) Also when we use undefined values except CCR, we generate Kripke structure.

We verify seven programs in the following experiment environment.

- Windows 8.1
- Intel (R) Core (TM) i3-2120T CPU @ 2.60GHz
- Available memory area : 2GB

Simulator is written in a combination of Java and Scala, and Model Checker is written in Java as follows.

- Java 1.7.0 45 , 15000 lines
- Scala 2.10.3 , 5000 lines
- tools : JFlex [18] Jacc [19]

Table 2 Verifying a model while generating the model

stack size(B)	state	relation	time(s)	stack overflow
1024	1398	1397	33.3	true
512	758	757	17	true
256	438	437	10.2	true
48	177	176	4.1	true

Table 3 Verifying a model after generating the model

stack size(B)	state	relation	time(s)	stack overflow
1024	-	-	-	Time Out
512	-	-	-	Time Out
254	92823	92822	6649.9	true
48	17683	17682	1889.3	true

4.2.2 Experiments

We show results of experiments in from Table 2 to Table 8. The items of each table consists of the number of states and relations, required time, stack overflow. Required time is total time of both Simulator and Model Checking. stack overflow shows stack overflow occurs or not (true/false).

1. In order to evaluate verifying a model by model checking while generating the model by dynamic program analysis, we show Table 2 and Table 3. Here true means that stack overflow occurs, and Time Out means that a result is not given in 24 hours. By comparing Table 2 and Table 3, verifying a model by model checking while generating the model by dynamic program analysis is very effective.

2. In order to evaluate undefined values, we show Table 4, Table 5 and Table 6.

    a. When we do not use undefined values for all, we must refine seven 32bit registers in an initial state. For this reason, we can not get a result for the state space explosion as shown in Table 5.

    When we use undefined values for all, we can verify programs except PID and Linetrace as shown in Table 4. Whenever AD conversion is carried out by PID program, 2^8 states are generated and causes the state explosion. Whenever a sensor inputs the external environment, eight states are generated with Linetarce program in addition to the problem of PID program.

    We show undefined values very effective as shown in Table 4 and Table 5.

Table 4 Using undefined values considering clock cycles

Program	states	relations	time(s)	stackoverflow
LED	26909	28613	523	false
PID	-	-	-	Time Out
Stack	177	176	4.2	true
Tsensor_LED	13664	14996	334.8	false
Tsensor_motor	14842	15054	599.8	false
Tsensor_P	106495	108883	7352.1	false
Linetarce	-	-	-	Time Out

Table 5 Without undefined values considering clock cycles

Program	states	relations	time(s)	stackoverflow
LED	-	-	-	OutofMemory
PID	-	-	-	OutofMemory
Stack	-	-	-	OutofMemory
Tsensor_LED	-	-	-	OutofMemory
Tsensor_motor	-	-	-	OutofMemory
Tsensor_P	-	-	-	OutofMemory
Linetarce	-	-	-	OutofMemory

Table 6 Using undefined values except CCR considering clock cycles

Software	state	relation	time(s)	stackoverflow
LED	107709	1145444	2474.1	false
PID	-	-	-	Time Out
Stack	194	193	5.3	true
Tsensor_LED	54713	60056	1307.5	false
Tsensor_motor	60357	61504	2735.9	false
Tsensor_P	-	-	-	Time Out
Linetarce	-	-	-	Time Out

Table 7 Using undefined values without clock cycles

Software	state	relation	time(s)	stackoverflow
LED	-	-	-	OutofMemory
PID	-	-	-	OutofMemory
Stack	177	176	4.1	true
Tsensor_LED	-	-	-	OutofMemory
Tsensor_motor	-	-	-	OutofMemory
Tsensor_P	-	-	-	OutofMemory
Linetarce	-	-	-	OutofMemory

b. As shown in Table 4 and Table 6, the number of states in the case of using undefined values except CCR increases to approximately 4 times than the number of states in the case of using undefined values. As CCR is a special register, we evaluate undefined values of CCR. Using undefined values of CCR is slightly effective.

3. In order to evaluate considering clock cycles, we show Table 7, Table 8 and Table 9. When we do not consider clock cycles, we can not verify programs except Stack program even if we use undefined values for all. When we do not consider clock cycles, an interrupt is carried out disorderly. Therefore the state spece explosion occurs.

   Here we denote stack overflow by stackoverflow, and denote Out of Memory by OutofMemory.

Our proposed verification system has the following originality: (1) generating models including clock cycles, (2) abstract and refinement method of the bit level, (3) generating exact models, (4) verifying a model by model checking while generating it by dynamic program analysis.

We show the above techniques such as (1), (2) and (4) very effective by our experiments.

Table 8 Without undefined values without clock cycles

Software	state	relation	time(s)	stackoverflow
LED	-	-	-	OutofMemory
PID	-	-	-	OutofMemory
Stack	-	-	-	OutofMemory
Tsensor_LED	-	-	-	OutofMemory
Tsensor_motor	-	-	-	OutofMemory
Tsensor_P	-	-	-	OutofMemory
Linetarce	-	-	-	OutofMemory

Table 9 Using undefined values execept CCR without clock cycles

Software	state	relation	time(s)	stackoverflow
LED	-	-	-	OutofMemory
PID	-	-	-	OutofMemory
Stack	194	193	5.3	true
Tsensor_LED	-	-	-	OutofMemory
Tsensor_motor	-	-	-	OutofMemory
Tsensor_P	-	-	-	OutofMemory
Linetarce	-	-	-	OutofMemory

5. Conclusion

In this paper, we explain verifying embedded assembly programs. We generate the exact models including clock cycles, and develop abstract and refinement method of the bit level by undefined values. Also we verify Kripke structure by model checking while generating the Kripke structure by dynamic program analysis. Our proposed verification system has the following originality: (1) generating models including clock cycles, (2) abstract and refinement method of the bit level, (3) generating exact models, (4) verifying a model by model checking while generating it by dynamic program analysis. We show the above techniques very effective by our experiments.

In the future, we will verify embedded assembly programs based on CEGAR(Counterexample-guided abstraction refinement). We will verify liveness properties by extending our proposed method.

References

[1] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking, MIT Press, 1999.
[2] R. Jhana and R. Majumdar, "Software model checking," ACM Comput. Surv., vol.41, no.4, 2009.
[3] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," LNCS 4963, pp.337–340, 2008.
[4] E.M. Clarke, E.A. Emerson, and J. Sifakis, "Model Checking: Algorithmic Verification and Debugging," Commun, ACM, vol.52, no.11, pp.74–84, 2009.
[5] Corporation, R.E., Renesas Electronics, Renesas Electronics Corporation (online), http://japan.renesas.com/, 2014.
[6] nuvo WHEEL:ZMP, http://www.zmp.co.jp/products/wheel, 2016.
[7] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," International Journal on Software Tools for

Technology Transfer, vol.9, no.5-6, pp.505–525, 2007.
[8] G. Weissenbacher, The BOOP Toolkit v0.42, Graz University of Technology (online), available from http://boop.sourceforge.net/ (accessed 2014-6-17).
[9] B. Schlich and S. Kowalewski, "Model Checking C Source Code for Embedded Systems," International Journal on Software Tools for Technology Transfer, vol.11, no.3, pp.187–202, 2009.
[10] B. Schlich, "Model Checking of Software for Microcontrollers," ACM Trans. Embed. Comput. Syst., vol.9, no.4, pp.1–27, 2010.
[11] B. Schlich, J. Brauer, and S. Kowalewski, "Application of Static Analyses for State-space Reduction to the Microcontroller Binary Code," Sci. Comput. Program., vol.76, no.2, pp.100–118, 2011.
[12] T. Noll and B. Schlich, "Delayed Nondeterminism in Model Checking Embedded Systems Assembly Code," LNCS 4899, pp.185–201, 2008.
[13] G.J. Holzmann, "The Engineering of a Model Checker: The Gnu i-Protocol Case Study Revisited," LNCS 1680, pp.232–244, 1999.
[14] K. Yorav and O. Grumberg, "Static Analysis for StateSpace Reductions Preserving Temporal Logics," Form. Methods Syst. Des., vol.25, no.1, pp.67–96, 2004.
[15] L.I. Millett and T.Teitelbaum, "Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation," International Journal on Software Tools for Technology Transfer, vol.2, no.4, pp.343–349, 2000.
[16] M.C. Browne, E.M. Clarke, and O. Grümberg, "Characterizing Kripke Structures in Temporal Logic," LNCS 249, pp.256–270, 1987.
[17] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," ACM Trans. Program. Lang. Syst., vol.8, no.2, pp.244–263, 1986.
[18] G. Klein, JFlex - The Fast Scanner Generator for Java, CSE UNSW (online), available from (http://jflex.de/) (accessed 2014-6-27).
[19] M.P. Jones, Jacc: just another compiler compiler for Java, Department of Computer Science and Engineering at the OGI School of Science & Engineering at OHSU (online), available from (http://jflex.de/) (accessed 2014-6-27).

Satoshi Yamane received B.S., M.S and Ph.D. degrees from Kyoto Univeristy. Now he is a professor of Kanazawa University. He interested in formal verification and real-time and distributed computing.

Ryosuke Konoshita received M.S degree from Kanazawa University in 2014. Now he works in SUGINO.

Tomonori Kato received M.S degree from Kanazawa University in 2015. Now he works in software house.