Simulation and Model Checking of Embedded Assembly Program

Satoshi Yamane, Tomonori Kato, Ryosuke Konoshita
Kanazawa University

Abstract

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 including clock cycles by exhaustively and dynamically simulating assembly programs, and simultaneously verify it by model checking in order to avoid the state space explosion. In addition, we have introduced undefined values to reduce the number of states.

keyword: Embedded assembly program, Model checking, Simulation

1 Introduction

Recently software model checking [1] [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 the model. The reasons to verify assembly programs are as follows:

1. We realize program verification at the level of registers. From this, we can verify stack overflow and stack underflow.
2. We realize verifying timing errors. For this, we 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 structure by dynamic program analysis. We verify whether stack overflow or stack underflow occurs or not by our proposed run-time exhaustive verification. In order to avoid the state space explosion, we propose the following methods.

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 quite 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 the structure to be dangerous when it is safe.

© 2016 Information Processing Society of Japan

4. As we verify Kripke structure by model checking while generating it 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, Section 2 introduces Kripke structure and model checking. Our proposed verification system is described in Section 3. Experiments of embedded robot software are described in Section 4. Finally, Section 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 such as BLAST[7], BOOP[8] and Schlich' Model Checker[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 over-approximated model by static program analysis, and then verifies it by model 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 when replacing abstract values with concrete values.

In this paper, our proposed method is quite different from [MC]SQUARE as follows: (1)Generating models including clock cycles and computing the execution time, (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 it 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 quite different from Lynette Millett's method.

Our previous work [16] simulatse assembly program, and verifies whether it reaches bad states or not. This paper extends our previous work [16] with temporal logic model checking.

2 Kripke structure and model checking

We define Kripke structure [17] 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 Kripke structures [18]. 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.

© 2016 Information Processing Society of Japan

In this paper, we verify whether stack overflow happens or not. We specify stack overflow by CTL [18] as follows.

$AG(s_{STACK} \leq LIMIT_{STACK})$
$= \neg EF(s_{STACK} > LIMIT_{STACK})$

,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 Figure 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 corresponding assembly program, based on dynamic program analysis by exhaustive breadth-first search. 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 denotes input and output information from environments, events, registers. A state $s \in S$ is defined by values of registers, memory, stack pointer and program counter. The value of 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 $s.PC$.

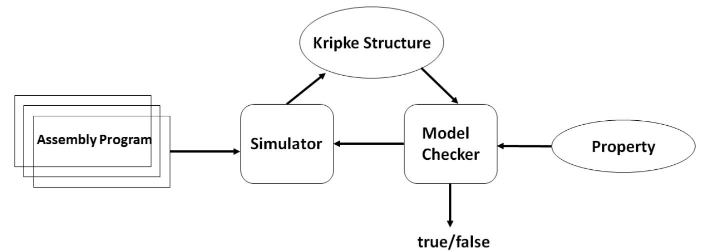

Figure 1: Configuration of verification system

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, by Simulator in Figure 1, in an initial state s_0, all enabled interruptions are executed by InterruptHandling, and then InterruptHandling generates successor 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 structure by model checking (line 47,50). We assume an interrupt processing is one instruction.

2. Next, by Simulator in Figure 1, 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).

3. Finally, by Model Checker and Property in Figure 1, ModelCheckEF verifies the Kripke structure by model checking (line 47,50). In this paper, Property is $EFformula$, and it is defined in line 1,2.

While a new state is generated, that is, while *list* is not empty, Algorithm 1 repeats the above procedure.

© 2016 Information Processing Society of Japan

But when EFf holds true in s_0, 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 interruption 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*. Also, we compute the execution time by the clock cycles of the instruction operation. But we do not consider delay on the architecture. 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), $EF\ f$ 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 Figure 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 $R(s, s')$.

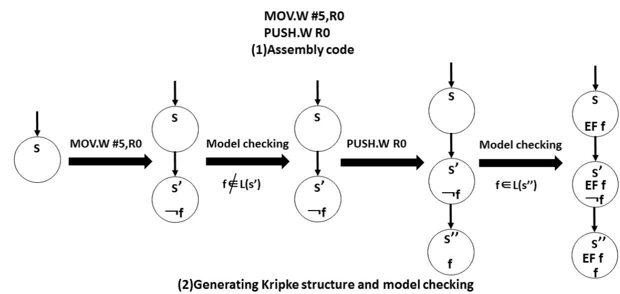

Figure 2: Example of Simulation and Model Checking

4 Experiments of verification system

4.1 Embedded software

The experiment of our verification system demonstrates the effects of our proposed techniques. We

© 2016 Information Processing Society of Japan

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)
11:        if decidable interrupts don't exist then
12:            ExecuteInstruction(s)
13:        end if
14:        if EFf ∈ L(s_0) then break
15:        end if
16:        remove s from list
17:    end while
18:    if EFf ∈ L(s_0) then return (S, R, true)
19:    else return (S, R, false)
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'
27:            PC_i = VectorTable[i]
28:            s'.PC = PC_i               ▷ set PC_i to PC of s'
29:            GlobalMaskBit_s' ← true    ▷ mask s'
30:            InterruptFlag_s' ← false   ▷ clear flag
31:            AddNewState(s, s')
32:            ExecuteInstruction(s')
33:        end if
34:    end for
35: end function
36:
37: function ExecuteInstruction(s)
38:    operation ← memory[s.PC]
39:    s' ← execute(s, operation)
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')}
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
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
```

used seven programs written for H8/3687 microcontroller [5] [6]. 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 it by dynamic program analysis with verifying a model after generating it, using only stack program. When we verify a model by model checking while generating it by dynamic program analysis, we confirm

© 2016 Information Processing Society of Japan

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
Line-trace	249	811

it 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[19] Jacc[20]

Table 2: Verifying a model while generating it

stack size(Byte)	state	relation	time(s)	so
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 it

stack size(B)	state	relation	time(s)	so
1024	-	-	-	TO
512	-	-	-	TO
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). In the following tables, so means stack overflow, TO means Time Out, and OM means Out of Memory.

1. In order to evaluate verifying a model by model checking while generating it 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 it 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 Line-

ⓒ 2016 Information Processing Society of Japan

Table 4: Using undefined values considering clock cycles

Program	states	relations	time(s)	so
LED	26909	28613	523	false
PID	-	-	-	TO
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
Line-trace	-	-	-	TO

Table 5: Without undefined values considering clock cycles

Program	states	relations	time(s)	so
LED	-	-	-	OM
PID	-	-	-	OM
Stack	-	-	-	OM
Tsensor_LED	-	-	-	OM
Tsensor_motor	-	-	-	OM
Tsensor_P	-	-	-	OM
Line-trace	-	-	-	OM

Table 6: Using undefined values except CCR considering clock cycles

Software	state	relation	time(s)	so
LED	107709	1145444	2474.1	false
PID	-	-	-	TO
Stack	194	193	5.3	true
Tsensor_LED	54713	60056	1307.5	false
Tsensor_motor	60357	61504	2735.9	false
Tsensor_P	-	-	-	TO
Line-trace	-	-	-	TO

Table 7: Using undefined values without clock cycles

Software	state	relation	time(s)	so
LED	-	-	-	OM
PID	-	-	-	OM
Stack	177	176	4.1	true
Tsensor_LED	-	-	-	OM
Tsensor_motor	-	-	-	OM
Tsensor_P	-	-	-	OM
Line-trace	-	-	-	OM

trace 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 Line-trace program in addition to the problem of PID program.

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

(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. 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.

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.

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 it by dynamic program analysis. Our

ⓒ 2016 Information Processing Society of Japan

Table 8: Without undefined values without clock cycles

Software	state	relation	time(s)	so
LED	-	-	-	OM
PID	-	-	-	OM
Stack	-	-	-	OM
Tsensor_LED	-	-	-	OM
Tsensor_motor	-	-	-	OM
Tsensor_P	-	-	-	OM
Line-trace	-	-	-	OM

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.

Acknowledgments

We would like to thank anonymous referees for a number of useful comments. This work was partially supported by Kakenhi 15K00093.

References

[1] Clarke, E.M., Grumberg,O. and Peled,D.A: Model Checking, MIT Press (1999).

[2] Ranjit Jhana, Rupak Majumdar: Software model checking, ACM Comput. Surv. 41(4) (2009).

[3] Leonardo de Moura, Nikolaj Bjorner: Z3:An Efficient SMT Solver, LNCS 4963, pp.337-340 (2008).

[4] Clarke, E. M., Emerson, E. A. and Sifakis, J.: Model Checking: Algorithmic Verification and Debugging, Commun. ACM, 52(11), pp. 74-84 (2009).

[5] Corporation, R. E.: Renesas Electronics, Renesas Electronics Corporation(online), available from http://japan.renesas.com/ (2014).

[6] nuvo WHEEL:ZMP (http://www.zmp.co.jp/products/wheel) (2016).

[7] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar:The software model checker Blast. International Journal on Software Tools for Technology Transfer, 9(5-6), pp.505-525 (2007)

[8] Weissenbacher, G.: The BOOP Toolkit v0.42, Graz University of Technology (online), available from http://boop.sourceforge.net/(accessed 2014-6-17).

[9] Schlich, B. and Kowalewski, S.: Model Checking C Source Code for Embedded Systems, International Journal on Software Tools for Technology Transfer, 11(3), pp. 187-202 (2009).

[10] Schlich, B.: Model Checking of Software for Microcontrollers, ACM Trans. Embed. Comput. Syst., 9(4), pp. 1-27 (2010).

[11] Schlich, B., Brauer, J. and Kowalewski, S.: Application of Static Analyses for State-space Reduction to the Microcontroller Binary Code, Sci. Comput. Program., 76(2), pp. 100-118 (2011).

[12] Noll, T. and Schlich, B.: Delayed Nondeterminism in Model Checking Embedded Systems Assembly Code, LNCS 4899, pp. 185-201 (2008).

[13] Holzmann, G. J.: The Engineering of a Model Checker: The Gnu i-Protocol Case Study Revisited, LNCS 1680, pp. 232-244 (1999).

[14] Yorav, K. and Grumberg, O.: Static Analysis for StateSpace Reductions Preserving Temporal Logics, Form. Methods Syst. Des., 25(1), pp. 67-96 (2004).

[15] Lynette I. Millett and Tim Teitelbaum : Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation, International Journal on Software Tools for Technology Transfer, 2(4), pp.343-349 (2000)

ⓒ 2016 Information Processing Society of Japan

[16] Konoshita R., Yamane S., Sakurai K.:Model Checking by Modeling for Embedded Assembly Programs, Embedded Systems Symposium 2014, pp.13-21 (2014).

[17] Browne, M. C., Clarke, E. M. and Grumberg, O.: Characterizing Kripke Structures in Temporal Logic, LNCS 249, pp. 256-270 (1987).

[18] Clarke, E.M. and Emerson, E.A. and Sistla, A.P:Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. Program. Lang. Syst. 8(2), pp.244-263 (1986)

[19] Klein, G.: JFlex - The Fast Scanner Generator for Java, CSE UNSW (online), available from ihttp://jflex.de/(accessed 2014-6-27).

[20] Jones, M. P.: 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).

ⓒ 2016 Information Processing Society of Japan