TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 10/20 pp684-690
Volume 12, Number 6, December 2007

Verification of Concurrent Assembly Programs with a Petri Net Based Safety Policy*

WANG Shengyuan (王生原)**, LIANG Yingyi (梁英毅), DONG Yuan (董 渊)

Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

Abstract: Concurrent programs written in a machine level language are being used in many areas but verification of such programs brings new challenges to the programming language community. Most of the studies in the literature on verifying the safety properties of concurrent programs are for high-level languages, specifications, or calculi. Therefore, more studies are needed on concurrency verification for machine level language programs. This paper describes a framework of a Petri net based safety policy for the verification of concurrent assembly programs, to exploit the capability of Petri nets in concurrency modeling. The concurrency safety properties can be considered separately using the net structure and by mixing Hoare logic and computational tree logic. Therefore, more useful higher-level safety properties can be specified and verified.

Key words: verification; machine level concurrent programming; Petri nets; safety policy; verifying/certifying compilers

Introduction

There are many reasons to construct concurrent programs in a machine level language. First of all, translation of the concurrent structure from a high-level source code/specification to a machine-level code can improve the capability to statically check safety properties. A concurrent machine level code also enhances the possibility of improving the object code quality. Concurrent machine level programs will become more common in core system libraries, especially with more multi-thread and multi-core architecture in the future. Therefore, the verification of concurrent machine level programs is an important topic in the research community.

As a grand challenge problem[1] to the programming language community, the development of a verifying/certifying compiler can serve as a good framework to statically check the safety properties of a program.

One pioneering work in the area is the PCC by Necula and Lee[2,3]. Touchstone[2] and SpecialJ[4] are examples of PCC based prototypes. Other examples include TAL[5,6], ECC[7], JFlow[8], CAP[9], and CCAP[10].

Safety policy is one key element in the design of verifying/certifying compilers. In the literature, most well developed safety policies are based on type safety. However, type safety is not sufficient for verifying the concurrent machine level programs.

Yu, Hamid, and Shao developed a logic-based approach for verifying assembly code by allowing semi-automatic proof, with the verified properties being more than type safety[9]. Recently, they extended their work to verify the safety properties for concurrent assembly code[10]. However, the application of this

Received: 2007-01-22; revised: 2007-06-30

* Supported by the Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList), the National Natural Science Foundation of China (No. 60573017), and the National High-Tech Research and Development (863) Program of China (No. 2006AA01Z198)

** To whom correspondence should be addressed.
E-mail: wwssyy@tsinghua.edu.cn; Tel: 86-10-62794240

approach is limited due to the complexity of Hoare logic[11] formulae to specify the safety properties of a concurrent assembly program.

The safety properties on concurrent programs written in high-level languages, specifications, or calculi are well studied in the literature, with methods such as CSP[12], π-Calculus[13], Petri nets[14], and UNITY[15]. Additional work is needed to extend these methods to machine level languages.

This paper describes a framework of a Petri net based safety policy for the verification of concurrent assembly programs, to exploit the capability of Petri nets in concurrency modeling. In the approach, the concurrency is considered with a net structure and the execution of the atomic instruction sequences initiated by the transition firing. The safety policy mixes Hoare logic for specifying the behavior of the atomic instruction sequences and computational tree logic[16] for specifying the concurrent behavior. After a preliminary analysis, the existing tool Coq[17] can serve as the proof assistant with INA[18] as the model checking assistant.

1 Background

Figure 1 shows a simple verifying/certifying framework. The verifying/certifying information produced in the compile-time is packaged together with the executable code, with the information downloaded at the same time the executable code is downloaded by the code consumer. If the package passes the verifier check, the executable code should be safe to run.

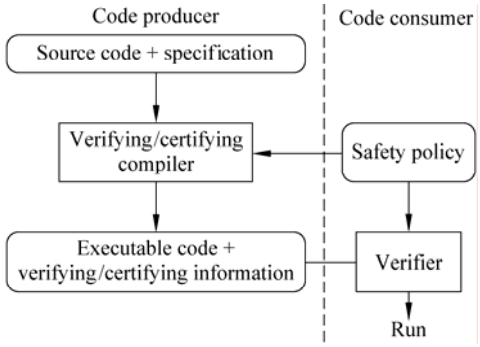

Fig. 1 A simple verifying/certifying framework

Verifying/certifying information can be any information to certify the safety properties of the executable code, such as type, proof, and even model checking. Both proof and model checking information are used in this project.

Safety policy can be implemented in different levels, corresponding to the different levels of verifiable safety properties, from some basic safety properties, such as protecting the system against crashes (such as safety for control flows, memory, and stacks), to more sophisticated ones (such as liveness).

The framework of safety policy in this project is based on Petri nets. Hoare logic formulae (or Hoare expressions) are used for specifying the safety property of atomic instruction sequences. Some basic safety properties (such as deadlock free) and computational tree logic (CTL) formulae are used for specifying the safety properties of the Petri net system.

The safety policy design takes several aspects into account:

- The concurrency safety properties can be easily specified, verified, analyzed, and visualized in the Petri nets.
- Localizing the logic inference to sequential codes can reduce the work load for producing the proof information and will result in a positive effect in general, though the model checking information should be added.
- The Petri net based approach can be smoothly associated with our recent work. We are developing Petri net based machine descriptions at both the thread-level and the instruction-level, which are specific to the compiler design. The Petri net model in this paper can be regarded as a thread-level virtual machine model specific to safety verification. The virtual machine model can be connected with the real target machine description through thread-level (even instruction-level when necessary) scheduling algorithms to optimize the object code. Some static information for the virtual machine model can be extracted automatically from the real machine descriptions in the process.

2 Technical Considerations for Safety Policy

Technically, safety policy should consider the following aspects:

- **Machine-level programming model** (machine model, in short). An execution environment for machine-level programs and operational semantics for the machine-level language need to be defined. The

framework of safety policy in this paper uses a thread-level virtual machine model based on an extended colored Petri net. For simplicity, the Petri net model is not defined formally in this paper.

- **Definable and verifiable safety properties**. One or more logic systems should be defined for the machine model, with a tradeoff between the expressiveness and verifiability. The extent of automation of production of the verifying/certifying information is also an important factor to be considered.

2.1 Thread-level virtual machine model

A thread-level virtual machine model, TVM, is defined based on an extended colored Petri net[19] model with the following characteristics:

- The defined color sets include Register, Location, InstructionSequence, and NullColor.
- A color in Register is identified by a single RegisterID. Similarly, a color in Location is identified by a single LocationID, and a color in InstructionSequence is identified by a single InstructionSequenceID.
- A color identified by a RegisterID has one attribute holding the current value of the corresponding register. Similarly, a color identified by a LocationID has one attribute holding the current value of the corresponding memory location, and a color identified by an InstructionSequenceID has one attribute holding an atomic instruction sequence.

A TVM system is a TVM net together with an initial marking.

The operational semantic of a TVM system can be defined similarly as usual colored Petri net systems, except for the following modifications:

- While an InstructionSequenceID is bound, the enabling condition depends additionally on whether or not all the required resources for the execution of the corresponding atomic instruction sequence are also bounded at the same time. Here, the required resources consist of registers and locations to be accessed during the execution. Therefore, the systems requires an associate structural analysis capability.
- While an InstructionSequenceID is bound, the result of firing the transition is additionally determined by the operational effect of the corresponding atomic instruction sequence, which can be obtained from semantic functions in these instructions.

The semantic function of each instruction is constructed according to how it affects the machine state of the TVM system. The machine state describes the current content of the registers, locations, and identified instruction sequences related to different threads.

The programming model and the exact syntax and semantics of an assembly-level language must be determined to give a complete formalization for the machine model TVM. In this paper, only a few RISC-like instructions are taken as examples with a normal programming model, so related formalization is not included but only the safety policy is introduced.

2.2 Safety properties

The safety properties in the Petri net based safety policy can be classified as follows:

- A Hoare expression is used to specify the safety property of each atomic instruction sequence. Some auxiliary variables may be needed in the Hoare expression. Existing proof assistants, such as Coq[17], can be used to mechanically produce the proof[9,10].
- The concurrency safety properties can be separately considered with the help of CTL logic. The CTL logic is suitable for verifying finite-state concurrent systems. The Petri net tool INA[18] can be used to specify the properties in the form of the CTL formula and for model checking. For example, one can easily describe concurrency, mutual exclusion, and progressiveness by using the CTL formula and then checking them interactively. The CTL formula corresponding to place/transition color bindings (Registers, Locations, and InstructionSequences) can be used to get the pre-/post-predicate for each atomic instruction sequence to form a Hoare expression. The proof process for each of these Hoare expressions can then be integrated into the model checking procedure.
- Most of the dynamic safety properties in a TVM system are very useful and can be analyzed using existing Petri net tools. For example, INA and CPN tools[20] can easily analyze many behavioral properties, such as boundedness and liveness.

3 Example

This section illustrates some features of the Petri net-based safety policy by a simple example.

Figure 2 shows a colored Petri net system specifying the behavior of a dining philosophers system[18]. A concurrent assembly program using the colored Petri net system is built to get a TVM system. The colored Petri net system can be regarded as a concurrent control structure of the concurrent assembly program. The functional aspect of the program is fulfilled by instruction sequences bound to the transition colors, and the interactions among these instruction sequences are through the shared registers and/or memory locations bound to the place colors.

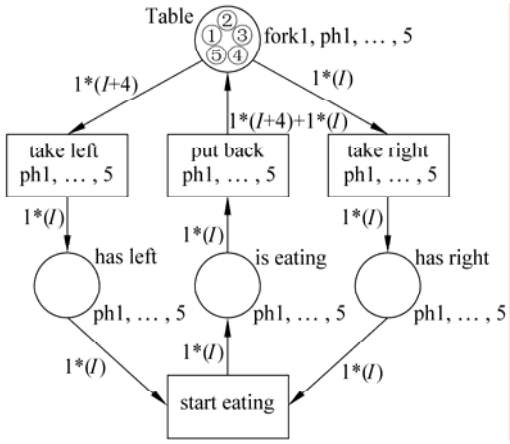

Fig. 2 Dining philosophers

In Fig. 3, *M*(fork_base), *M*(fork_base+1), ⋯, and *M*(fork_base+4) stand for 5 memory locations with the base fork_base and offsets 0 to 4. Each *M*(fork_base+*i*) is bound to a place color table.fork*i*+1, where table is a place depicted in Fig. 2. Similarly, each of the 5 memory locations, *M*(ph_base+*i*), is bound to 3 place colors has_left.ph*i*+1, has_right.ph*i*+1 and is_eating.ph*i*+1.

In Fig. 4, the instruction sequences take_left*i*'s are bound to the transition colors take_left.ph*i*'s, the instruction sequences take_right*i*'s are bound to the transition colors take_right.ph*i*'s, the instruction sequences start_eat*i*'s are bound to the transition colors start_eat.ph*i*'s, and the instruction sequences put_back*i*'s are bound to the transition colors put_back.ph*i*'s.

The following observations can help explain the binding relations: (1) Instruction sequence take_left1 can access the shared memory location *M*(fork_base+4) if and only if the token fork5 in the place table can be won over. Notice that *M*(fork_base+4) is bound to table.fork5. (2) Instruction sequence start_eat1 can access the shared memory location *M*(ph_base) if and only if the token ph1's in both place has_left and place has_right are available. Notice that *M*(ph_base) is bound to both has_left.ph1 and has_right.ph1.

M(fork_base)	<=>	table.fork1
M(fork_base+1)	<=>	table.fork2
M(fork_base+2)	<=>	table.fork3
M(fork_base+3)	<=>	table.fork4
M(fork_base+4)	<=>	table.fork5
M(ph_base)	<=>	has_left.ph1, has_right.ph1, is_eating.ph1
M(ph_base+1)	<=>	has_left.ph2, has_right.ph2, is_eating.ph2
M(ph_base+2)	<=>	has_left.ph3, has_right.ph3, is_eating.ph3
M(ph_base+3)	<=>	has_left.ph4, has_right.ph4, is_eating.ph4
M(ph_base+4)	<=>	has_left.ph5, has_right.ph5, is_eating.ph5

Fig. 3 Place color bindings

take_left1	<=>	take_left.ph1
take_left2	<=>	take_left.ph2
take_left3	<=>	take_left.ph3
take_left4	<=>	take_left.ph4
take_left5	<=>	take_left.ph5
take_right1	<=>	take_right.ph1
take_right2	<=>	take_right.ph2
take_right3	<=>	take_right.ph3
take_right4	<=>	take_right.ph4
take_right5	<=>	take_right.ph5
start_eating1	<=>	start_eating.ph1
start_eating2	<=>	start_eating.ph2
start_eating3	<=>	start_eating.ph3
start_eating4	<=>	start_eating.ph4
start_eating5	<=>	start_eating.ph5
put_back1	<=>	put_back.ph1
put_back2	<=>	put_back.ph2
put_back3	<=>	put_back.ph3
put_back4	<=>	put_back.ph4
put_back5	<=>	put_back.ph5

Fig. 4 Transition color bindings

Figure 5 defines the initial values, invariants, and assertions for the shared memory locations.

Initial values:

M(fork_base)=0
M(fork_base+1)=0
M(fork_base+2)=0
M(fork_base+4)=0
/* 0—free, 1—being held by philosopher1 */
/* 2—being held by philosopher2; 3—being held by philosopher3 */
/* 4—being held by philosopher4; 5—being held by philosopher5 */
M(ph_base)=0
M(ph_base+1)=0
M(ph_base+2)=0
M(ph_base+3)=0
M(ph_base+4)=0
/* 0—thinking; 1—eating; 2—waiting */

Invariants:

M(fork_base)=0∨M(fork_base)=1∨M(fork_base)=2
…
M(fork_base+4)=0∨M(fork_base+4)=5∨M(fork_base)=1
M(ph_base+2)=0∨M(ph_base+2)=1∨M(ph_base+2)=21
…
M(ph_base+2)≠1∨(M(ph_pase+1)≠1∧M(ph_base+3)≠1)
…

Asserts:

P_i: ∀j:0≤j≤4∧j+1≠i, M(ph_base+j)=a_j
Tl_i: ∀j:0≤j≤4∧j+1≠(i+1) mod 5, M(fork_base+j)a_j
TR_i: ∀j:0≤j≤4∧j+1≠i, M(fork_base+j)=a_j
S_i: ∀j:0≤j≤4∧j+1≠i∧j+1≠(i+1) mod 5, M(fork_base+j)=a_j

Fig. 5 Initial values, invariants, and asserts

Invariants specify some global safety properties about the concurrent assembly program, which must be true for the whole run time. For example, the following invariant states that philosopher 2 and philosopher 4 cannot be eating when philosopher 3 is eating:

$$M(\text{ph\_base+2}) \neq 1 \vee (M(\text{ph\_base}+1) \neq 1 \wedge M(\text{ph\_base}+3) \neq 1).$$

The asserts sections define assertions used in the logical specification of instruction sequences.

Figure 6 shows the instruction sequences and their logical specifications with the pre-conditions and post-conditions. Figure 6 is for the instruction sequences take_left1, take_right1, start_eat1, and put_back1. The specifications for other instruction sequences are similar.

take_left1: -{M(fork_base+1)=0; M(ph_base)=0; TL_1; P_1}
-{consumes table.fork2}
mov r1, fork_base
movi r2, 1
st 1(r1), r2
mov r1, ph_base
movi r2, 2
st 0(r1), r2
end take_left1: -{M(fork_base+1)=1; M(ph_base)=2; TL_1; P_1}
-{produces has_left.ph1}
take_right1: -{M(fork_base)=0; M(ph_base)=0; TR_1; P_1}
-{consumes table.fork1}
mov r1, fork_base
movi r2, 1
st 0(r1), r2
mov r1, ph_base
movi r2, 2
st 0(r1), r2
end take_right1: -{M(fork_base+1)=1; M(ph_base)=1; TR_1; P_1}
-{produces has_right.ph1}
start_eating1: -{M(fork_base+1)=1; M(fork_base)=1; M(ph_base)=2; S_1; P_1}
-{consumes has_left.ph1, has_right.ph1}
mov r1, ph_base
movi r2, 1
st 0(r1), r2
end start_eating1: -{M(fork_base+1)=1; M(fork_base)=1; M(ph_base)=1; S_1; P_1}
-{produces is_eating.ph1}
put_back1: -{M(fork_base+1)=1; M(fork_base)=1; M(ph_base)=1; S_1; P_1}
-{consumes is_eating.ph1}
mov r1, ph_base
movi r2, 0
st 0(r1), r2
mov r1, fork_base
movi r2, 0
st 0(r1), r2
st 1(r1), r2
end put_back1: -{M(fork_base+1)=0; M(fork_base)=0; M(ph_base)=0; S_1; P_1}
-{produces table. fork1, table. fork2}

Fig. 6 Instruction sequences with logical specifications

In the example, the semantic for instruction "movi rt, immediate" is to move an immediate to register rt, the semantic for instruction "mov rt, loc" is to move the content of memory location loc to register rt, and the

semantic for instruction "st r*t*, offset(base)" is to move the content of register r*t* to memory location *M*(base+ offset).

The pre-condition of an instruction sequence is a conjunction of predicates that relate to shared registers or memory locations. The post-conditions are similar.

The annotations such as consumes table.fork2 and produces has_left.ph1 are not requisite, but are helpful in the construction of the verifier.

An instruction sequence together with its associate pre-condition and post-condition forms a Hoare expression, which is used for specifying the safety property of each atomic instruction sequence. Some of the global safety properties are specified through invariants, such as those in Fig. 5. The proof information is then produced from these specifications and the operational semantics of the concurrent assembly language, with the help of proof assistants.

The concurrency safety properties are considered separately based on the colored Petri net. First, the behavioral properties of the net are analyzed using Petri net tools. For example, an INA analysis showed that the colored Petri net in Fig. 2 has the following properties: the net is structurally bounded; the net is bounded; number of dead states found: 2; the net has dead reachable states; the net is not live; the net is not live and safe; the net is not live, if dead transitions are ignored; and the net is safe.

More flexible safety properties can be specified by the CTL logic formula and these properties can be verified using the INA tool. For example, for the net in Fig. 2, the check result for formula "AX-P3" is "The formula is TRUE".

Here, AX is a temporal-logical quantor meaning "always" and P3 is a predicate equivalent to

is_eating(Ph3) ∧ is_eating(Ph2) ∨ is_eating(Ph3) ∧ is_eating(Ph4).

AX-P3 states that P3 is not true for all future time, that is, it is not possible for both Philosopher 2 and Philosopher 4 to be eating while Philosopher 3 is eating. The AX-P3 checking procedure can be produced by INA.

The project seeks to extract enough model checking information from the checking procedure and other useful trace information to create pre-/post-conditions related to the procedure for each associate instruction sequence. Then the model checking procedure is produced mechanically for the concurrent assembly program. The Hoare expressions will be formed automatically from the CTL formula and the proof process for each of these Hoare expressions will be integrated into the model checking procedure.

4 Concluding Remarks

This paper describes a framework of a Petri net based safety policy for the verification of concurrent assembly-level programs. The concurrency safety properties can be considered separately using the net structure and by mixing Hoare logic and computational tree logic, so that more useful higher-level safety properties can be specified and verified. In our verifying/certifying compiler projects, the model checking procedure can be used for verifying/certifying information.

Future work will verify that proof and model checking information can be successfully produced and integrated into a verifier.

References

[1] Hoare T. The verifying compiler: A grand challenge for computing research. In: Proc. 2003 International Conference on Compiler Construction (CC'03), LNCS Vol. 2622. Warsaw, Poland: Springer-Verlag, 2003: 262-272.

[2] Necula G C. Compiling with proofs [Ph. D. Dissertation]. Carnegie Mellon University, USA, 1998.

[3] Necula G C, Lee P. The design and implementation of a certifying compiler. In: Proc. Conf. Programming Language Design and Implementation, ACM SIGPLAN. Montreal, Quebec, Canada, 1998: 333-344.

[4] Colby C, Lee P, Necula G C. A certifying compiler for Java. In: Proc. Conf. Programming Language Design and Implementation, ACM SIGPLAN. Vancouver, British Columbia, Canada, 2000: 95-107.

[5] Morrisett G, Crary K, Glew N, Walker D. Stack-based typed assembly language. In: Leroy X, Ohori A, eds. Proc. Workshop on Types in Compilation, Kyoto, Japan, LNCS Vol. 1473. Berlin, Germany: Springer-Verlag, 1998: 28-52.

[6] Morrisett G, Crary K, Glew N, Grossman D, Samuels R, Smith F, Walker D, Weirich S, Zdancewic S. TALx86: A realistic typed assembly language. In: Proc. Workshop on Compiler Support for System Software, ACM SIGPLAN. Atlanta, GA, USA, 1999: 25-35.

[7] Kozen D. Efficient code certification. Technical Report

98-1661. Computer Science Department, Cornell University, USA, 1998.

[8] Myers A C. Jflow: Practical static information flow control. In: Proc. 16th Symp. Principles of Programming Languages, ACM. San Antonio, Texas, USA, 1999: 228-241.

[9] Yu Dachuan, Hamid N A, Shao Zhong. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 2004, **50**(1-3): 101-127.

[10] Yu Dachuan, Shao Zhong. Verification of safety properties for concurrent assembly code. In: Proceedings of ICFP'04. Snowbird, Utah, USA, 2004: 19-22.

[11] Hoare C A R. An axiomatic basis for computer programming. *Communications of the ACM*, 1969, **12**(10): 576-580.

[12] Hoare C A R. Communicating sequential processes. *Communications of the ACM*, 1978, **21**(8): 666-677.

[13] Milner R. Communicating and Mobile Systems: The π-Calculus. Cambridge, UK: Cambridge University Press, 1999.

[14] Resig W. Petri Nets: An Introduction, Vol. 4 of EATCS Monographs on Theoretical Computer Science. Berlin, Germany: Springer-Verlag, 1985.

[15] Chandy K M, Misra J. Parallel Program Design: A Foundation. Boston, MA, USA: Addision-Wesley Publishing Company, 1988.

[16] Clarke E M, Emerson E A, Sistla J. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems*, 1986, **8**(20): 244-263.

[17] The Coq Development Team. The Coq proof assistant reference manual. The Coq release v7.1, Oct. 2001.

[18] Roch S, Starke P H. INA: Integrated Net Analyzer Version 2.2 Manual. Department of Computer Science, Humboldt-University of Berlin, April 1999. http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina.

[19] Jensen K. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1, EATCS Mono graphs in Computer Science. Berlin Heideberg and New York: Springer-Verlag, 1992.

[20] CPN Tools Version 2.2.0. http://daimi.au.dk/CPNToals, 2007-3-A.

Three Tsinghua Research Groups to Receive National Natural Science Foundation Support

The National Natural Science Foundation China (NSFC) announced on September 27 that three Tsinghua research groups will receive the Science Fund for Creative Research Group support in 2007.

These three research groups, led respectively by academician Xue Qikun from the Department of Physics, Professor Luo Jianbin from the Department of Precision Instruments and Mechanology, and Professor Zhou Donghua from the Department of Automation, will each receive five million RMB over a three-year period.

In order to provide steady support for frontier research efforts in basic science and to promote prominent young scientists and their research groups with innovative capabilities, the NSFC launched the Science Fund for Creative Research Group pilot program in 2000.

（http://news.tsinghua.edu.cn）