I.J. Wireless and Microwave Technologies, **2011, 3, 62-69**
Published Online June 2011 in MECS (http://www.mecs-press.net)
DOI: 10.5815/ijwmt.2011.03.10

Available online at http://www.mecs-press.net/ijwmt

The Study on Formal Verification of OS Kernel

Zhang Yu[a], Dong Yunwei[b], Zhang Zhongqiu[c], Huo Hong[d], Zhang Fan[e]

[a,b,c,d,e]*School of Computer Science, Northwestern Polytechnical University, Xi'an, China*

Abstract

There is increasing pressure on providing a high degree of assurance of operation system's security and functionality. Formal verification is the only known way to guarantee that a system is free of programming errors. We study on formal verification of operation system kernel in system implementation level and take theorem proving and model checking as the main technical methods to resolve the key techniques of verifying operation system kernel in C implementation level. We present a case study to the verification of real-world C systems code derived from an implementation of μC/OS – II in the end.

Index Terms: OS kernel; formal verification; theorem proving; model checking

1. Introduction

There is increasing pressure on providing a high degree of assurance of a computer system's safety and functionality. In manufacturing, defense, traffic, aviation, spaceflight, critical infrastructure control, automotive systems, medical service, assisted living and other key application domain, there have been lots of tremendous losses caused by the system failure which the core control software design flaws brought about. As accidents caused by software error or failure are becoming more and more, people come to realize that the system under the condition of high complexity, conventional software engineering methods and software design, evaluation method cannot solve embedded software reliability and safety design problems in depth. This calls for end-to-end guarantees of systems functionality, from applications down to hardware.

While safety certification is increasingly required at higher system levels, the operating system is generally confidence to be safe. This clearly presents a weak link in today's safety-critical systems, given the size and complexity of modern operating systems，which exhibits an increasing number of bugs. Worse still, numerous fixes, distributed by their vendors, may introduce new errors or render other system components inoperative.

The only real solution to establish trustworthiness is formal verification, proving the implementation correct. It's about explicit and strict mathematical proofs of the correctness of a system. This has, until recently, been considered to be an intractable proposition — the OS layer was too large and complex to poorly scale formal

*Corresponding author.
E-mail address: [a]yuzhang.nwpu@gmail.com; [b]yunweidong@nwpu.edu.cn; [c]seven39@126.com; [d]huohong@mail.nwpu.edu.cn; [e]zhangfan@ nwpu.edu.cn

methods. However, there is a renewed tendency towards smaller OS kernels means that the size of the program to be verified is only around 10,000 loc. It is possible to use formal verification instead of traditional methods for this area. In this paper we study on formal verification of this smaller OS kernel in system implementation level, which is a weak link in the trustworthy of OS kernel and is related to system eventually correctness.

The next section provides an overview of OS verification and its application to kernels. Section 3 gives a more detailed two different formal verification methods for C program. Section 4 present a case study to the verification of real-world C systems code derived from an implementation of $\mu C/OS-\mathrm{II}$, which not only provides an opportunity to validate the models against realistic code, but also allows us to compare and contrast the two methods in practice. Section 5 summarizes our work and prospect.

2. OS Verification

To get an impression of current industry best practice, we look through the software assurance standard: RTCA/DO-178B[1] and Common Criteria[2]. RTCA/DO-178B is an industry-accepted guidance for satisfying airworthiness requirements which provides guidelines for the production of software for airborne systems and equipment. Systems are categorized by DO-178B as meeting safety assurance levels A through E based on their criticality in supporting safe aircraft flight. And Common Criteria is a standard for software verification that is mutually recognized by a large number of countries. It textual research software level from the methodological perspective and the software artefacts are: the software requirements, the functional specification, the high-level design of the system, the low-level design, and finally the implementation. There are seven levels of assurance (EAL 1–7) in the standard, which generate partitions by the treatment of each software artefact. None of currently commercially available OS kernels has been formally verified.

Recently, NICTA from Australia has made an OS verification project named L4.verified[3]. The project is providing a mathematical, machine-checked proof of the functional correctness of the seL4 microkernel with respect to a high level, formal description of its expected behavior. And the aim is to produce a truly trustworthy, high-performance operating system kernel. They think starting the verification directly from the C source without any higher-level specification should be expected to be a difficult and long process. In contrast to the OS approach, the traditional formal methods approach would take the design ideas, formalize them into a specification first and then analyze that specification. Based on this, C-level implementation verification only needs to verify function correctness.

Formal verification can reduce the larger gap between user requirements and implementation and hence gain increased confidence in system correctness. It makes others convince that the implementation of software fulfils its specification. Therefore, system correctness is described by means of a formal method, then the standard procedure through certain validation rules of these formalization specifications and relevant code verification, judge whether the program in accordance with the procedure specification indicated by the way of implementation.

In program verification field, predicate abstract method[4] presented by Graf is a kind of program oriented model abstract methods, which abstract program into finite state machine model based on a set of limited quantity predicate and then can use model-checking tool to verify. Combined with CEGAR （Counter-Example Guided Abstraction Refinement）method, model establishment and verification methods based on predicate abstract can verify software source code automatically. PCC (Proof - Carrying code)[5] and FPCC (Foundational Proof - Carrying code) [6]based on logical method, through carrying the proof of source codes, provides a mechanism that guarantee the safety of code before run. But because lack of type expression ability the, PCC itself will only verify program’s simple attributes such as type safe. CAP[7] makes the program in the most general attributes can be verified by improving PCC expression. It is program verification method based on Hoare logic style in the assembly level.

3. Verification Method

Takes theorem proving and model-checking as the main technical methods to resolve the key techniques of verifying OS microkernel. The details are as follows.

3.1. Ttheorem Proving

We adopt program correctness validation technology based on Hoare logic[8] to establish the axiom semantics of C program. And then, use Coq as an interactive theorem proving tool to prove program correctness.

Hoare logic provides a formal system for reasoning about program correctness. Hoare logic is based on the idea of a specification as a contract between the implementation of a function and its clients. The specification is made up of a pre-condition and a post-condition. The pre-condition is a predicate describing the condition the function relies on for correct operation; the client must fulfill this condition. The post-condition is a predicate describing the condition the function establishes after correctly running; the client can rely on this condition being true after the call to the function. Hoare logic uses Hoare Triples to reason about program correctness. A Hoare Triple is of the form [P]S[Q] or {P}S{Q}, where P is the pre-condition, Q is the post-condition, and S is the statement(s) that implement the function.

Definition 1(termination): if each input a that makes P(a) true, program S will terminate, said the program S is terminated to P. Use Sterminate to stand for it.

Definition 2(partially correct): if S is executed in a store initially satisfying P and it terminates, then the final store satisfies Q. Use [P]S[Q] to stand for it. Partially correct form: $[P]S[Q]\,\text{iff}\ (\forall a)(P(a)\text{and}(\text{Sterminate})) \rightarrow Q$

Definition 3(totally correct): assuming the P is satisfied before S executes, the S is guaranteed to terminate and when it does, the post-condition satisfies Q. Thus total correctness is partial correctness in addition to termination. Use {P}S{Q} to stand for it. Totally correct form: $\{P\}S\{Q\}\,\text{iff}\ (\forall a)(P(a) \rightarrow ((\text{Sterminate})\,\text{and}\,Q)$

And some rules of Hoare logic are as follows:

skip： $\{P\}\text{skip}\{P\}$

assign： $\{P[x \mapsto a]\}x := a\{P\}$

sequence： $\dfrac{\{P\}S_1\{Q\},\{Q\}S_2\{R\}}{\{P\}S_1;\ S_2\{R\}}$

if： $\dfrac{\{b \wedge P\}S_1\{Q\},\{\neg b \wedge P\}S_2\{Q\}}{\{P\}\text{if}\ (b)\ \text{then}\ (S_1)\ \text{else}\ (S_2)\{Q\}}$

while： $\dfrac{\{b \wedge P\}S\{P\}}{\{P\}\text{while}\ (b)\ \text{do}\ (S)\{\neg b \wedge P\}}$

cons: $$\frac{P \to P_1, \{P\}S\{Q\},\ Q_1 \to Q}{\{P_1\}S\{Q_1\}}$$

The main steps of program verification include: program designers provides additional properly assert for program, and then generates verification conditions and theorem prove assistant completes the proof of verification conditions. Here are the main steps in detail.

- Provide proper assertion

The automatic generation of loop invariants is still an unsolved problem. It refers to solving a fixed point of recursive formula, and solving this equation is usually undecidable. And a useful loop invariant precisely expressed relationship between variables which are operated by looping statements in program body. However, searching for an effective loop invariant is full of challenges. Therefore, the source code needs the programmer to provide the appropriate assertions, including the entrance of function, the exit of function and loop invariant.

- Generate verification conditions

Hoare logic constructs a contract between the implementation of a function and its clients. But search of its pre-condition is very difficult. Therefore, we use the weakest pre-predicate logic to calculus the pre-condition in Hoare logic. In this way, we can get pre-condition mechanically. The weakest pre-condition calculus[9] is proposed by Dijkstra which is used to perform program correctness proof and reason about the program. Its basic idea is in order to verify $\{P\}S\{Q\}$ we need to find out all P' called Pre(S, Q), which make $\{P'\}S\{Q\}$ established. Verify that $(\exists P')\ P' \in \text{Pre}(S, Q)$, $P \Rightarrow P'$. In these P', looking for a weakest pre-condition, take it as the pre-condition of the program. Therefore proof process becomes to calculate WP(S, Q), and prove $P \Rightarrow \text{WP}(S, Q)$. Here are the rules:

WP(skip,Q)=Q

WP("x = E", Q) = Q[E/x]

WP("S1;S2" , Q) = WP (S1, WP(S2, Q))

WP(IF B {S1} else {S2}, Q) = $(B \Rightarrow \text{WP}(S1, Q)) \wedge (\neg B \Rightarrow \text{WP}(S2, Q))$

WP(while B {C }, Q) = I $\wedge$ ($I \wedge B \Rightarrow \text{VC}(C, I)$) $\wedge$ $(I \wedge \neg B \Rightarrow Q)$, I is loop invariant, B is loop condition.

- Interactive theorem proving

Theorem proving method with high abstractions can process infinite state system theoretically. We use high order logic to describe the system and the system properties. Then transfer properties to be verified into theorem described by mathematical logic. In the end, use theorem proof assistant Coq[10] with axioms, proved theorem and reasoning rules to verify specification is correct in high order logic system.

3.2. Model Checking

Model checking works on a model of the system that is typically reduced to what is relevant to the specific properties of interest. The model checker then exhaustively explores the model's reachable state space to determine whether the properties are held. It is an automatic verification method, and can provide counterexample path when some properties are not satisfied. The general model checking tools required to use their special modeling language. So when you use these tools, you must abstract system model manually.

Model checking is only feasible for systems with a moderately-sized state space, which implies dramatic simplification. Hence, this approach usually does not give guarantees about the actual system.

Therefore, direct model checking for software program is based on model abstraction, which abstracts the finite state space model from program. Based on the predicate abstract, modeling and verification of source code can be automatic. BLAST [11] is a model checking tool for C program, which developed by Berkeley California university. This tool is based on a counterexample automatically abstract refinement technology to construct abstraction model. It uses lazy predicate abstraction and interpolation-based predicate discovery methods to abstract , verify and refine the state space of program. This tool can not only verify security attributes of sequence C program, but also verify concurrent C program. And use theorem proof assistant Simplify to solve abstract state transition relationship. Model checking has been applied to the OS layer and has shown utility here as a means of bug discovery in code involving concurrency. So we try to use BLAST to verify OS kernel.

4. Case Study

In this paper we present a case study in the application of our models to the verification of real-world C systems code (Os_Core.c) derived from an implementation of $\mu C/OS-II$. Function is shown in Fig.1:

```
#define  OS_ENTER_CRITICAL()  asm  CLI
#define  OS_EXIT_CRITICAL()   asm  STI
……
void  OSSchedLock (void){
#if OS_CRITICAL_METHOD == 3
   OS_CPU_SR  cpu_sr;
#endif
       if (OSRunning == TRUE) {
      OS_ENTER_CRITICAL();
      if (OSLockNesting < 255) {
         OSLockNesting++; }
      OS_EXIT_CRITICAL();
 }}
```

Fig 1. C source code.

This function is used to prevent rescheduling to take place. This allows your application to prevent context switches until you are ready to permit context switching. $\mu C/OS-II$ define two macros to deal with interrupt switch: OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL(). When access critical sections, must use OS_ENTER_CRITICAL() to open interrupt, then use OS_EXIT_CRITICAL() before leave critical sections. This mechanism has three different implementations. In some special hardware, first implementation is the only choice. So we take this implementation.

4.1. Theorem Proving Approach

According to theorem proving approach, program designers provide additional proper assertion for the program. Then generates verification conditions and theorem proof assistant completes the proof of verification conditions. Here we analysis reasoning process in Coq. This process is shown in Fig.2.

```
Fixpoint equal_0(v:Var):nat:=match {(v)} with C 0=>1|
                             other => 0 end.
Definition mutex_pre(mutex : Var):Eprop:= fun E =>
E mutex=0.
Definition mutex_post(mutex : Var):Eprop:= fun E =>
E mutex=1 \/ E mutex=0.
Definition mutex_prog(mutex: Var):=
br( C(equal_0 mutex))(mutex <- (C 1))nil.
Theorem mutex_ok:
correct(mutex_prog VX)(mutex_pre VX)(mutex_post VX).
Proof.
```

Fig 2. definition in Coq.

For mutex_ok theorem, we prove it with Hoare logic reasoning and rich strategy libraries provided by Coq. We should prove every objectives generated by each proof step. When all targets have been proved, mutex_ok will be proved successfully. Details are shown in Fig.3.

```
apply okBr.
 apply okAssign.
 unfold mutex_pre in |- *.
 unfold upd in |- *.
 simpl in |- *.
 intuition.

 unfold mutex_pre, mutex_post in |- *.
 unfold upd in |- *.
 simpl in |- *.
 intuition.
 apply
  okConseq
   with (fun E : Env => E VX = 0 /\ 0 = 0) (fun E : Env => E
VX = 0 /\ 0 = 0).
  auto.

  intuition.

  apply okNil.

mutex_ok is defined
```

Fig 3. proven strategy.

4.2. Model Checking Approach

In order to detect the program sequence security attributes, we usually need to add the corresponding observation variables and statements in the program to get to observe the value of the variable. In this case, the global variable Mutex is used to mark whether OS_ENTER_CRITICAL () or OS_EXIT_CRITICAL () is used alternately. BLAST uses relatively independent code description language to detect the sequence security attributes, which can protect the integrity of the source code as far as possible. Fig.4 shows the specification document.

```
global int Mutex= 0;
event {
pattern { OS_ENTER_CRITICAL(); }
guard {Mutex== 0 }
action {Mutex= 1; }}
event {
pattern { OS_EXIT_CRITICAL();}
guard { Mutex== 1 }
action { Mutex = 0; }}
```

Fig 4. OS_ENTER_CRITICAL().spc.

According to the concept of mutually exclusive, continuous twice to lock or unlock critical area is impracticable. When the program is invoked, OS_ENTER_CRITICAL (twice) or OS_EXIT_CRITICAL () function will trigger the Mutex variables and then trigger ERROR tags. After this, use BLAST command to check the program. When BLAST finishes the check, it returns the result. The result of the souce code is that the system is safe. Fig. 5 shows the result.

```
Mutex==0
Mutex ==1
Read 2 predicates
Begin Building CFA ……
Finished Building CFA ……
addPred: 0: (gui) adding predicate Mutex ==0 to the system
addPred: 1: (gui) adding predicate Mutex ==1 to the system
Forking Simplify process ……
            No error found.  The system is safe :-)
```

Fig 5. check result.

5. Conclutions

We have presented our experience in formally verifying OS kernel. The challenges for formal verification at the kernel level relate to performance, size, and the level of abstraction. Since the early attempts at kernel verification there have been dramatic improvements in the power of available tools. Tools like Coq and BLAST have been used in a number of successful verifications. This has led to a significant reduction in the cost of formal verification, and a lowering of the feasibility threshold. At the same time the potential benefits have increased

We take theorem proving and model-checking as the main technical methods to resolve the key techniques of verifying OS kernel. We have shown that full, rigorous, formal verification is practically achievable for OS kernels with very reasonable effort compared to traditional development methods.

Acknowledgment

This paper is supported by the National Natural Science Foundation of China under Grant No.60736017 and Basic Research Foundation of Northwestern Polytechnical University under Grant No.JC200917.

References

[1] RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification[S]. Requirements and Technical Concepts for Aviation (RTCA), Dec. ,1992
[2] US National Institute of Standards. Common Criteria or IT Security Evaluation, 1999. ISO Standard 15408. http://www.niap-ccevs.org/cc-scheme/
[3] Gerwin Klein, June Andronick, Kevin Elphinstone, et al. seL4: Formal verification of an OS kernel. Communications of the ACM, 53(6), 107–115, (June, 2010).
[4] S.Graf and H.Saidi. Construction of abstract state graphs with PVS. CAV 97: Computer-aided Verification, LNCS 1254 , 72-83. 1997.
[5] Necula1 G. Proof-carrying code [C]. In: Proc of the 24th ACM SIGPLAN-SIGACT Symp on Principles of Programming Language (POPL'97) .New York : ACM Press , 1997.106 -119
[6] Apple A W. Foundational proof-carrying code[C]. Proceedings of 16th Annual IEEE Symposium on Logic in Computer Science. Baston, Massachusetts. USA,2001:247-258
[7] Dachuan Yu , N A Hamid , Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation [J]. Science of Computer Program , 2004 , 50 (1-3) : 101 – 127
[8] C A R Hoare. An axiomatic basis for computer programming [J].Communications of the ACM,1969;12(10):576-580
[9] Yanfang Ren, Jing Yang, Bingrui Suo,Checking method based on program correctness, Computer Engineering and Design. 2009,30 (17)(in Chinese)
[10] Y Bertot, P Casteran. Coq'Art:The Calculus of Inductive Constructions[M].Berlin: Springer- Verlag, 2004.
[11] Thomas A.Henzinger, Ranjit Jhala, Rupak Majumdar and Gregoire Sutre,Software Verification with BLAST. 10th Int SPIN Workshop (SPIN'2003).