Formal design and verification of system task in intelligent transportation systems based on micro-kernel architecture

Zhenjiang Qian¹ · Yong Jin¹ · Gaofei Sun¹ · Xiaoshuang Xing¹ · Kaijian Xia²

Received: 22 December 2020 / Accepted: 17 August 2021
© The Author(s) 2021

Abstract

The accuracy of design and implementation of an operating system in intelligent transportation systems is difficult to describe and validate because of its complexity. In this paper, we describe an OS in intelligent transportation systems with automaton theory and establish an OS state model. Based on this model, we construct an isomorphic model in Isabelle/HOL, describe the work objects and operational semantics of the system, and verify the system at the assembly level. We use a micro-kernel OS prototype (VSOS) for intelligent transportation systems as an example to illustrate our method and verify the correctness of design and implementation in VSOS with Isabelle/HOL. Verification shows that the proposed method is feasible.

Keywords State automaton model · System task · Formal design · Formal verification · Theorem proving

1 Introduction

Operating system (OS) in intelligent transportation systems (ITS) is the foundation and platform for running other software. The safety and reliability of an OS in ITS is a primary topic of information security.

Due to the complexity of the OS in ITS, traditional methods, e.g., system testing, cannot completely solve security problems because many defects are difficult to find.

According to the test plan, software testing uses functional testing tools to perform functional tests on software products to discover software errors and evaluate software quality. Software testing is widely used in industry because of its easy operation and high efficiency. It is undeniable that software testing does play a decisive role in reducing software errors and improving software quality. However, there are certain deficiencies in software testing. It can find software errors, but it cannot explain that the software has no errors. Software errors have not ever completely disappeared for software testing. Errors in some software systems still cause major problems.

The software source includes the design and implementation of the software. Formal verification of correctness of the software source program is an important method to achieve high reliability of the software, and also in smart environments Corno and Sanaullah (2014).

Formal verification method at the source level of the software include dynamic detection and static analysis. Dynamic detection makes it possible to find defects and vulnerabilities in the running process of the source program by adding relevant detection code fragments in the source program. These methods such as dynamic monitoring increase the operating cost of software. At the same time, due to the uncertainty of the program running process, such as multi-core concurrency, parallelism, lock mechanism, etc., dynamic monitoring cannot completely guarantee the correctness of the source program. With static analysis method through strict analysis of the source program, you can find and diagnose the defects and vulnerabilities in the software before running, so as to guide and feedback to the software design.

Currently, the formal method to design and verify an OS in ITS is to use logical language to describe the semantics of the OS in ITS and verify that the operational semantics of system implementation meet predetermined functional requirements.

Static analysis of the software source program can be carried out from two aspects, including high-level (such as C language) formal verification and the underlying assembly level formal verification. Many scholars have conducted

✉ Zhenjiang Qian
qianzj@csig.edu.cn

¹ School of Computer Science and Engineering, Changshu Institute of Technology, Suzhou 215500, China

² Changshu Hospital of Soochow University, Jiangsu 215500, Changshu, China

research on formal verification at the high-level. However, the high-level language code is not machine code that actually runs on the machine. Even if the code at the high-level layer is formally verified, it can only show that the system is reliable at the high-level layer, but it cannot guarantee that the system actually runs correctly on the physical machine. The reason is that the assembly language layer is separated between the high-level language layer and the machine code layer. The transformation from the high-level language code to the assembly language code is realized by the compiler. To explain the consistency of the high-level language code and the assembly language code, there are two ways: the first is to explain the correctness of the compiler; the second is to model the assembly language code layer and verify the correctness of its semantics. For the first approach, due to the huge scale and complexity of the compiler, it is often difficult to verify its correctness Leroy (2009). For large-scale software systems, such as OS, in terms of manpower and material resources, many software development projects are prohibitive. The second method is to directly model and verify the assembly language layer. Because the assembly language is too low-level, it is difficult to formalize the verification. How to effectively model the assembly language code to facilitate its semantics verification of the correctness of the efficacy has become a major challenge in the formal field of OS.

Academia is very concerned about methods for formal verification of the safety of an OS in ITS, which is related not only to implementation but also, initially, to system design Desnitsky and Kotenko (2016). An OS in ITS designed without consideration of safety requirements may not be safe even though the implementation is correct. Therefore, at the design level, we need to consider issues of structure and functionality of the system and whether the functionality can meet security objectives.

System implementation is another issue impacting the reliability and accuracy of the system. The primary objective is to ensure consistency of the machine code, design functions and modules.

2 Related work

The first formal work on OS was UCLA Secure Unix Walker et al. (1980) and PSOS (Provably Secure Operating System) Feiertag and Neumann (1979) in the 1970s. Due to the limitations of programming languages and verification tools, formal projects during this period were limited to the verification of partial implementations of OS.

Early in this century, scholars and research groups continued formal work on OS in ITS. The Verisoft Alkassar et al. (2009) project designed and verified OS accuracy and functionality in ITS. Verisoft emphasized the formal verification

not only of the OS but also of the software stack from the underlying hardware to the upper applications. Although Verisoft did not reach their goals, the project demonstrated the possibility of complete verification of the system.

The NICTA seL4 group led by Klein G performed research on formalization of micro-kernel seL4 Heiser and Elphinstone (2016; Klein et al. 2014; Shapiro et al. 2004) with extensive results Daum et al. (2014); Elphinstone and Heiser (2013); Heiser et al. (2012). seL4 used Haskell to describe the system specification Klein et al. (2010). Haskell is similar to theorem prover Isabelle/HOL Nipkow et al. (2002) and can easily be converted to the input of Isabelle/HOL. To simplify the complexity of verification, seL4 finished formal verification of seL4 on a single-core platform. Recently, seL4 has attempted to design, implement and verify the system on a multi-core platform Heiser et al. (2020; Klein et al. 2018).

The Yale Flint team led by Professor Shao cooperated with Professor Chen of USTC to set up a USTC-Yale joint research centre for high-confidence software. Flint created a new logic programming language known as VeriML Stampoulis (2012) with a strong formal description and safety, and studied verification methods of programs and systems in a concurrent environment Wang et al. (2019); Gu et al. (2018; 2019). Flint used separation logic to verify the accuracy of memory sharing and data abstraction (Koenig and Shao 2018; Gu et al. 2015). Meanwhile, virtual timeline was used to specify and reason for preemptive scheduling, and a novel compositional framework is proposed for reasoning Liu et al. (2020). For distributed systems, a system called WormSpace was design and verified to supply an address space of the Write-Once Register Shin et al. (2019). Flint developed a programming language DeepSEA for specification and abstraction refinement of system software (?). Flint also studied the formal schedulability analysis for OS kernel Guo et al. (2019).

The team led by Professor Feng studied the compilation and formal verification for concurrent programs Jiang et al. (2019), partial methods for concurrent objects Liang and Feng (2018), and verification for kernel preemption Xu et al. (20116). A program logic LiLi was proposed for verifying linearizability and progress for concurrent objects Liang and Feng (2016).

In this paper, we contend that in order to guarantee the accuracy and safety of a system, we need to summarize the conditions for accuracy and safety using requirement analysis and system modelling. Based on these conditions, we design system functionality, propose targeted design principles and solutions, confirm that the system design does not violate the conditions, and verify that the system code conforms to the system design.

We describe the OS in ITS using automaton theory, and establish an OS state model. Based on this model, we

construct an isomorphic model in Isabelle/HOL, describe the work objects and operational semantics of the system, and verify the system at the assembly level. We use a micro-kernel OS prototype (VSOS) example to illustrate our method and verify the accuracy of the design and implementation of VSOS with Isabelle/HOL.

The rest of this paper is organized as follows. Section 3 describes the VSOS architecture. We establish the state automaton model in Sect. 4. Section 5 uses the system module-system task of VSOS as an example to illustrate the proposed formal verification method. Section 6 shows the evaluation of verification in our work. We conclude the paper and talk about future work in Sect. 7.

3 VSOS architecture

VSOS is our self-implemented micro-kernel OS prototype for ITS. VSOS consists of a micro-kernel running in kernel mode and other components such as service processes, third-party modules and drivers. Figure 1 depicts the overall architecture of VSOS.

VSOS provides support for a multi-core environment in which one micro-kernel can run on each physical core. VSOS micro-kernel handles the interrupt, exception, process scheduling, messaging, and clock and system tasks. The clock task handles the clock interrupt. The system task is a crucial task that runs in kernel mode and provides various privileged services for service processes, third-party modules and drivers running in user mode. In VSOS, there are separate address spaces for service processes, third-party modules and drivers, and these spaces are isolated from each other. The service processes, third-party modules and drivers can communicate only by messaging. Because of the isolation mechanism, processes running in user mode cannot access and modify the address space of micro-kernel but can only send service requests to the system task to indirectly complete data operations in micro-kernel space.

The micro-kernel, service processes, third-party modules and drivers are all message-driven in VSOS. The system task provides a variety of kernel-privileged operations for service processes, third-party modules and drivers running in user mode. VSOS utilizes an access control policy to ensure that the upper programs cannot send messages directly to the system task; the system task refuses these messages. The service processes, third-party modules and drivers are separated from the upper application processes to ensure the safety of the system.

In VSOS, the system task provides 32 kernel calls to service processes, third-party modules and drivers. The kernel calls can be divided into four functional categories: (1) privilege-required operations, e.g., *do_devio* needs kernel privileges to read and write I/O port; (2) cross-address space operations, e.g., *do_copy* copies data from one process to another process; (3) system core data operations, e.g., *do_fork* creates a new process by adding a new item in the process table; (4) operations to obtain kernel information, e.g., *do_getinfo* provides kernel information for service processes, third-party modules and drivers.

The service processes, third-party modules and drivers perform functional management using services provided by the system task and provide their respective services to the upper application processes.

4 State automaton model

To specify and verify the system, we establish a VSOS state automaton model and describe the system using automaton theory. The state automaton model describes the essence of the running system, and the states in the model can be described in the theorem prover Isabelle/HOL. Since the accuracy requirements of the system correspond to the logic formulae in Isabelle/HOL, we can verify the correctness of the system rigorously within the logical environment of Isabelle/HOL. In this section, we illustrate the VSOS state

Fig. 1 VSOS Architecture

automaton model and the method of constructing this model in Isabelle/HOL.

4.1 VSOS state automaton model

The VSOS state automaton is a quintuple, expressed as $M = (S, \Sigma, \delta, s_0, F)$. We divide the quintuple into three categories: the states (including state set S , initial state s_0 , and ending state F), the alphabet Σ and state transition function set δ . The correspondence between these elements of automaton and VSOS semantic elements are illustrated as follows:

- 1) state set S S represents the system state space. s_0 is the initial state of the system, and F is the end state. For VSOS, the set of all possible values of its operational work objects constitutes a Cartesian product space or a subset of this space. We define this set as the state space of VSOS. Through assignment to work objects, VSOS transforms the system state to a newly designated state to achieve the functionality of system services.
- 2) alphabet Σ Σ represents the automaton alphabet. Events during system operation constitute the automation alphabet, e.g., system calls, interrupts and messages.
- 3) state transition function set δ δ represents the state transition function set $S \times \Sigma \rightarrow S$. δ describes the state transition for different events. For example, suppose the event e_i occurs in the system state s_j , the system behaviour, e.g., a handler of system calls, interrupt and message processing, corresponds to a state transition function f . f operates on the work objects and transforms the system state to a new one s_k , expressed as $f : s_j \rightarrow s_k$.

For verification, we use HOL triples Hoare (1969), such as $\{P\}C\{Q\}$, to describe the accuracy of state transition functions where P is pre-conditions, Q is post-conditions, and C is the sequence of instructions. The triples mean that if the system state before execution of C meets pre-condition P , the state after execution of C will meet post-condition Q , i.e., $\forall s_j, s_k \in S. P s_j \wedge s_k = f(s_j) \rightarrow Q s_k$, where f is the corresponding state transition function of C .

4.2 Construction of VSOS state automaton model in Isabelle/HOL

To rigorously verify the correctness of design and implementation of VSOS, we construct the corresponding state model in Isabelle/HOL.

Isabelle is an interactive theorem prover and supports rich description capacity and safety types. We can use Isabelle/HOL to describe the specifications and execution of the software system. Isabelle/HOL represents the support and implementation of high-order logic (HOL) in Isabelle.

The verification process of Isabelle/HOL is expressed in the form of a theorem file. A theorem file is a set of types, functions, and theorem as shown in Fig. 2, where T is the theorem file name, and $B_1 \dots B_n$ are the theorem libraries loaded by T . The keywords **theory**, **imports**, **begin**, and **end** are internal labels of Isabelle/HOL. The components of *declarations*, *definitions*, and *proofs* are types, terms, and formulas.

Isabelle/HOL is a type system, and the main types include:

- 1) Basic types Such as bool type, natural number type, and integer type.
- 2) Construction type Such as list type, set type, compound type, e.g., “**int list**” means integer list. The label **record** can be used to construct record type, and the members of record type could be different types. The label **datatype** is used to customize new data types;
- 3) Function type Represented by “ $\Rightarrow$ ”, such as “**int $\Rightarrow$ nat**” represents a function mapping type from integer to natural number. Applying the function to the parameters forms the term. For example, assuming that f is a function of type “ $g \Rightarrow h$ ” and x is an item of type g , then “ fx ” is an item of type h .
- 4) Type variables Expressed as ‘ a ’, ‘ b ’, etc. For example, “‘ a **list**” means a list whose members’ type is ‘ a ’.

In Isabelle/HOL, the theorems are expressed in the form of formulas, and the process of proving the theorems is to prove the formulas as true.

Each function implemented in VSOS can be divided into two parts: work objects and instruction sequences. To describe work objects and instruction sequences in Isabelle/HOL, we construct a processor model from the aspect of assembly level of VSOS.

The assembly instruction is the smallest executable unit that operates its own work objects and causes a transition of the system state. The semantics of assembly instructions correspond to state transition functions. We defined the semantic functions for the most commonly used instructions of the X86 platform, e.g., mov, push, sub, add and

```
theory T

imports B1 ... Bn

begin

declarations, definitions, and proofs

end
```

Fig. 2 A theorem file in Isabelle/HOL

imp. In Isabelle/HOL, the instruction type of the VSOS state automaton module is defined as follows.

```
datatype vsos_instruction =
  addl int instr_type
    word32 reg word32 reg |
  subl int instr_type
    word32 reg word32 reg |
  jmpc int jmp_condition word32 |
  movl int instr_type
    word32 word32 reg
    reg word32 word32 reg reg |
  .....
```

datatype in Isabelle/HOL defines new data types. In the definition of *vsos_instruction*, each constructor corresponds to the actual assembly instruction.

According to the description of the system state, we define the system state as follows.

```
record vsos_state =
  C_vsos :: "word32  $\Rightarrow$  vsos_instruction"
  D_vsos :: "word32  $\Rightarrow$  byte"
  R_vsos :: regs
  msgs :: msg list
  call_nr :: nat
  .....
```

record defines the record type containing member types. The definition of *vsos_state* contains the code space *C_vsos* to describe the system code object with function mapping from memory space address (word32) to instruction (*vsos_instruction*); the data space *D_vsos* to describe the system data objects with function mapping from memory space address (word32) to data (byte); the register space *R_vsos* to describe the register objects; the message list *msgs* to describe the message list with the type of list of *msg*, where *msg* is the record type of message, including the source *m_src*, destination *m_dst*, and size *m_size* of the message; and the kernel call ID *call_nr* to describe the kernel call.

Before the end of each instruction execution, we use the auxiliary function *pc_add* to add the length of the current instruction to a program counter (PC register) so that the PC register points to the next instruction.

```
fun pc_add_vsos :: "vsos_state  $\Rightarrow$ 
  int  $\Rightarrow$  vsos_state" where
  "pc_add_vsos s n = s (| R_vsos :=
    ((R_vsos s)
    (| eip := (eip (R_vsos s)) +
    (word_of_int n) |) ) |)"
```

fun defines the total functions. For operational semantics of instructions, we define the function *step_vsos* : *vsos_state* $\Rightarrow$ *vsos_state*, where “*steps*” calculates the new state after execution of the instruction pointed to by the PC register in state *s*. For instruction sequences, we calculate the operational semantics through superposition of the execution effectiveness of multiple instructions. The corresponding function *stepN_vsos* is defined as follows.

```
primrec stepN_vsos :: "vsos_state  $\Rightarrow$ 
  nat  $\Rightarrow$  vsos_state" where
  "stepN_vsos s 0 = s" |
  "stepN_vsos s (Suc n) = stepN_vsos
    (step_vsos s) n"
```

primrec defines the primitive recursive functions.

5 Formal verification of system task in VSOS

Based on the formal description of the VSOS state automaton model in Sect. 3, this section illustrates the method to verify system accuracy at the assembly level. We use verification of the accuracy of system task initialization and kernel calls as examples.

5.1 Verification of accuracy of system task initialization

Before the system runs, the table of kernel calls should be correctly initialized. The system task initialization function *systask_init* stores the addresses of kernel call functions in a table of kernel calls. To ensure the accuracy of system task initialization, we guarantee that after the initialization of the table of kernel calls, the values of items in the table of kernel calls are equal to the expected ones. We define the predicate function *call_vec_right* as follows to check the accuracy of system task initialization:

definition *call_vec_right* :: “*vsos_state* $\Rightarrow$ *bool*” **where**
 “*call_vec_right* *s* = (
 ((*D_vsos* *s*) 0x1c0273adf) = 0 $\wedge$
 ((*D_vsos* *s*) 0x1c0273ae3) = 1 $\wedge$
 ((*D_vsos* *s*) 0x1c0273ae7) = 2 $\wedge$
 ((*D_vsos* *s*) 0x1c0273aeb) = 3 $\wedge$
 ...)”

Predicate function *call_vec_right* judges whether the values of the table of kernel calls is consistent with the expected values. We use the kernel call numbers to represent the values of items in the table of kernel calls. In the definition above, “(*D_vsos* *s*) 0x1c0273adf” represents the value at memory address 0x1c0273adf of data segment *D_vsos* in state *s*.

The pre-condition and post-condition of *systask_init* are defined as follows:

Pre-condition: for state *s* before *systask_init* runs the codes of *systask_init* should be in memory, and the pointer of process stack is set correctly. The definition of pre-condition is shown as *pre_cond_systask_init*.

Post-condition: state *s'* after *systask_init* runs meets the predicate function *call_vec_right*.

definition *pre_cond_systask_init* ::
 “*vsos_state* $\Rightarrow$ *bool*” **where**
 “*pre_cond_systask_init* *s* $\equiv$ (
 ((*C_vsos* *s*) 0 =
 pushl 1 regi 0 ebp) $\wedge$
 ((*C_vsos* *s*) 1 =
 movl 2 rr 0 0
 esp unused_r 0 0
 ebp unused_r) $\wedge$
 ((*C_vsos* *s*) 3 = subl 3 cr 0x28
 unuse_r 0 esp) $\wedge$
 (*initialize*)
 ((*C_vsos* *s*) 6 = movl 7 cm 0 0
 unuse_r unuse_r 0xc0273ae0 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 13 = movl 7 cm 1 0
 unuse_r unuse_r 0xc0273ae4 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 20 = movl 7 cm 2 0
 unuse_r unuse_r 0xc0273ae8 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 27 = movl 7 cm 3 0
 unuse_r unuse_r 0xc0273aec 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 34 = movl 7 cm 4 0
 unuse_r unuse_r 0xc0273af0 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 41 = movl 7 cm 5 0
 unuse_r unuse_r 0xc0273af4 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 48 = movl 7 cm 6 0
 unuse_r unuse_r 0xc0273af8 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 55 = movl 7 cm 7 0
 unuse_r unuse_r 0xc0273afc 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 62 = movl 7 cm 8 0
 unuse_r unuse_r 0xc0273b00 0
 unuse_r unuse_r) $\wedge$
 ((*C_vsos* *s*) 69 = movl 7 cm 9 0
 unuse_r unuse_r 0xc0273b04 0
 unuse_r unuse_r) $\wedge$

)”

The correctness theorem of *systask_init* is shown as follows.

Theorem *systask_init_correctness*:

$$\begin{aligned} &\forall s. \text{pre_cond_systask_init } s \wedge \\ &\quad (s' = \text{stepN_vsos } s \\ &\quad \quad \text{systask_init_steps}) \\ &\quad \rightarrow \text{call_vec_right } s' \end{aligned}$$

Theorem *systask_init_correctness* represents that for state s that meets condition *pre_cond_systask_init*, the new state s' after *systask_init* runs should meet condition *call_vec_right*.

For the proof of theorem *systask_init_correctness*, we calculate the operational semantics of each instruction by expanding the function *stepN_vsos* to get the end state after the system task initialization runs and then verify whether the end state meets *call_vec_right*. The verification of theorem *systask_init_correctness* in Isabelle/HOL is shown in Fig. 3.

5.2 Verification of accuracy of kernel calls

The system task provides kernel calls to service processes, third-party modules and drivers. In this section, we use the kernel call function *do_copy* as an example to illustrate our method of verification of the accuracy of kernel call functions. We use the predicate function as part of the precondition of the correctness theorem to define *do_copy* in Isabelle/HOL shown as *pre_cond_do_copy*.

definition *pre_cond_do_copy* ::
 “*vsos_state* $\Rightarrow$ *bool*” **where**
do_copy $s \equiv$ (
 ((*C_vsos* s) 10000 =
 pushl 1 *regi* 0 *ebp*) $\wedge$
 ((*C_vsos* s) 10001 =
 movl 2 *rr* 0 0 *esp* *unused_r* 0 0
 ebp *unused_r*) $\wedge$
 ((*C_vsos* s) 10003 =
 subl 3 *cr* 0x48 *unuse_r* 0 *esp*) $\wedge$
 ((*C_vsos* s) 10006 =
 movl 3 *mr* 0x8 0 *ebp* *unused_r*
 0 0 *eax* *unused_r*) $\wedge$
 ((*C_vsos* s) 10009 =
 movl 3 *mr* 0xc 0 *eax* *unused_r*
 0 0 *eax* *unused_r*) $\wedge$
 ((*C_vsos* s) 10012 =
 movl 3 *rm* 0 0 *eax* *unused_r*
 0xffffffff4 0 *ebp* *unused_r*) $\wedge$
 ((*C_vsos* s) 10015 =
 cmpl 7 *cm* 0x8ace *unused_r*
 0xffffffff4 *ebp*) $\wedge$
 ((*C_vsos* s) 10022 =
 jmpc 3 *jne* 10032) $\wedge$

 ”

The kernel call function *do_copy* provides the cross-address copy capability for service processes, third-party modules and drivers, and *do_copy* copies data of size bytes from address *src_ptr* of the source process to address *dst_ptr* of the destination process. When the source and destination processes are not the same, *do_copy* translates the address into a linear one in the kernel space.

The correctness theorem of *do_copy* is shown as follows.

Theorem *do_copy_correctness*:

$$\begin{aligned} &\forall s. \text{is_kernel_proc} \\ &\quad (m_src \text{ (hd (msgs } s))) \wedge \\ &\quad (\text{call_nr } s) = 15 \wedge \\ &\quad \text{call_vec_right } s \wedge \\ &\quad \text{parameter_valid } s \wedge \\ &\quad \text{pre_cond_do_copy } s \wedge \\ &\quad (s' = \text{stepN_vsos } s \text{ do_copy_steps}) \\ &\quad \rightarrow \text{CmpData } s' \text{ dst_ptr src_ptr size} \end{aligned}$$

Before *do_copy* runs, the system should meet the following conditions: (1) *do_copy* is called only by processes with kernel privileges, i.e., “*is_kernel_proc(m_src(hd(msgss)))* $\wedge$ (*call_nrs*) = 15 ”

Fig. 3 Verification of the correctness of system task initialization in Isabelle/HOL

shown in the theorem above where “ $hd(msgss)$ ” gets the head of the message list in state s . (2) The table of kernel calls remains unchanged satisfying the predicate $call_vec_right$. (3) The input parameters of do_copy should be checked by the auxiliary predicate $parameter_valid$ defined as follows.

definition $parameter_valid ::$
 $“vsos_state \Rightarrow bool”$ **where**
 $“parameter_valid\ s \equiv$
 $(D_vsos\ s\ (ebp\ (R_vsos\ s) + 20))$
 $=\ dst_ptr \wedge$
 $(D_vsos\ s\ (ebp\ (R_vsos\ s) + 12))$
 $=\ src_ptr \wedge$
 $(D_vsos\ s\ (ebp\ (R_vsos\ s) + 24))$
 $=\ size)$
 $”$

$parameter_valid$ compares the variables with the correct values of parameters in messages, and shows that the parameters of do_copy are correct when there is a valid comparison. In the definition above, “ $D_vsos(s, ebp(R_vsos(s)))$ ” represents the value at memory address “ $ebp(R_vsos(s))$ ” of data segment D_vsos in state s , and the subsequent constants, e.g., 20, 12, 24, represent offsets relative to the address.

After do_copy completes the data copy, the values of memory segment $[dst_ptr, dst_ptr + size)$ should be equal to the values of memory segment $[src_ptr, src_ptr + size)$. We define the following auxiliary function $CmpData$ to check the equality.

definition $CmpData ::$
 $“vsos_state \Rightarrow word32 \Rightarrow$
 $word32 \Rightarrow word32 \Rightarrow bool”$ **where**
 $“CmpData\ s\ p\ q\ n =$
 $(\forall\ i. (i \geq 0 \wedge i < n) \rightarrow$
 $(D_vsos\ s\ (p+i)) =$
 $(D_vsos\ s\ (q+i)))$
 $”$

$CmpData(s, p, q, n)$ represents the comparison of the values of n bytes in memory segments starting from address p and q in state s .

The verification of theorem $do_copy_correctness$ in Isabelle/HOL is shown in Fig. 4.

6 Evaluation of verification

For the whole verification of the system task of VSOS, the environment configured is: Dell OptiPlex XE 3 with I5-8500 3.0GHz of CPU, DDR4 8G of memory, Ubuntu 18.04, and Isabelle2020.

The count of the lines of code to verify the system task of VSOS in Isabelle/HOL is 14.7 k SLOC (Source Lines of Code), and that for the implementation (assembly code) is approximately 1.3 k SLOC as shown in Table 1, in which the correctness includes integrity and reliability. Judging from the ratio of the amount of implementation code by the one of verification code, the average is 1:10, and it shows that our method is feasible.

7 Conclusion

In this paper, we describe VSOS with automaton theory and establish VSOS state automaton. Based on this model, we construct an isomorphic model in Isabelle/HOL and verify the system at the assembly level. We use the system task of VSOS as an example to describe the work objects and operational semantics of the system using state automaton model and representing the correctness theorems of system functions with HOL triples. We verify the accuracy of design and implementation of the system task of VSOS in Isabelle/HOL. The verification results show that the proposed method is feasible.

Formal verification of an OS in ITS is difficult. Verification of the VSOS in Isabelle/HOL required 130.8 k SLOC, and implementation (assembly code) required approximately 11.4 k SLOC. Working with an OS in ITS is very labour-intensive and time-consuming. Therefore, how to further improve the efficiency of formal verification is the focus of future work. We plan to improve the process through modular verification and reuse.

Fig. 4 Verification of the correctness of kernel call do_copy in Isabelle/HOL

Table 1 Engineering quantities of verification of system task in ISABELLE/HOL

Verified module	Verified property	Verified code	Isabelle code	Verification workload
Privilege-required operations	correctness	0.27k SLOC	2.9	0.5 man-year
Cross-address space operations	correctness	0.49k SLOC	4.6	1.1 man-year
System core data operations	correctness	0.33k SLOC	3.7	0.9 man-year
Operations to obtain kernel information	correctness	0.21k SLOC	3.5	0.7 man-year

In the formal modeling and verification process of VSOS, for different modules, the verification logic used is different, such as higher-order logic, temporal logic and separation logic, etc. In the formal verification process of the whole system, how to explain that the whole system composed of various modules proved by different verification logics is complete, is also the focus of our next work.

It is worth mentioning that we establish the connection between VSOS and the logical system by constructing a formal model. The description of information such as hardware and objective environment in the real world is informal. The problem of isomorphism between the formal model and the real world requires strict formal verification. We plan to strictly formalize the description and verification for this problem of isomorphism from the perspective of requirement analysis, model theory and type theory.

Author contributions ZQ developed the verification framework and model in this study and drafted the manuscript. YJ, GS, XX and KX implemented the verification and result analysis.

Funding This work was supported in part by the Natural Science Foundation of Jiangsu Province under grant No. BK20191475, the fifth phase of “333 Project” scientific research funding project of Jiangsu Province in China under grant No. BRA2020306, the Qing Lan Project of Jiangsu Province in China under grant No. 2019.

Declarations

Conflicts of interest The authors declare no conflicts of interest.

Data availability The used data is available from the corresponding author upon request.

Code availability The code of this study is available from the corresponding author upon request.

Ethics approval Not applicable.

Consent to participate We consent to participate.

Consent for publication We consent for publication.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Alkassar E, Hillebrand MA, Leinenbach DC, Schirmer NW, Starostin A, Tsyban A (2009) Balancing the load: leveraging a semantics stack for systems verification. *J Autom Reason* 42(2–4):389–454
- Corno F, Sanaullah M (2014) Design-time formal verification for smart environments: an exploratory perspective. *J Ambient Intell Human Comput* 5(4):581–599
- Daum M, Billing N, Klein G (2014) Concerned with the unprivileged: User programs in kernel refinement. *Form Asp Comput* 26(5):1205–1229
- Desnitsky V, Kotenko I (2016) Automated design, verification and testing of secure systems with embedded devices based on elicitation of expert knowledge. *J Ambient Intell Human Comput* 7(4):705–719
- Elphinstone K, Heiser G (2013) From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, PA, USA, pp 133–150
- Feiertag RJ, Neumann PG (1979) The foundations of a provably secure operating system (PSOS). In: *Proceedings of the national computer conference*, Menlo Park, California, USA, pp 329–334
- Gu R, Koenig J, Ramananandro T, Shao Z, Wu N, Weng S, Zhang H, Guo Y (2015) Deep specifications and certified abstraction layers. In: *Proceedings of the 42nd ACM symposium on principles of programming languages*, Mumbai, India, pp 595–608
- Gu R, Shao Z, Kim J, Wu N, Koenig J, Sjöberg V, Chen H, Costanzo D, Ramananandro T (2018) Certified concurrent abstraction layers. In: *Proceedings of the ACM SIGPLAN conference on programming language design and implementation*, Philadelphia, PA, pp 646–661

- Gu R, Shao Z, Chen H, Kim J, Koenig J, Wu N, Sjöberg V, Costanzo D (2019) Building certified concurrent OS kernels. *Commun ACM* 62(10):89–99
- Guo X, Lesourd M, Liu M, Rieg L, Shao Z (2019) Integrating formal schedulability analysis into a verified os kernel. In: *Proceedings of the 31st international conference on computer aided verification*, New York, USA, pp 496–514
- Heiser G, Elphinstone K (2016) L4 microkernels: the lessons from 20 years of research and deployment. *ACM Trans Comput Syst* 34(1):1–29
- Heiser G, Murray T, Klein G (2012) It's time for trustworthy systems. *IEEE Secur Priv* 10(2):67–70
- Heiser G, Klein G, Andronick J (2020) seL4 in Australia: from research to real-world trustworthy systems. *Commun ACM* 63(4):72–75
- Hoare CAR (1969) An axiomatic basis for computer programming. *Commun ACM* 12(10):576–580
- Jiang H, Liang H, Xiao S, Zha J, Feng X (2019) Towards certified separate compilation for concurrent programs. In: *Proceedings of the ACM SIGPLAN conference on programming language design and implementation*, Phoenix, Arizona, USA, pp 111–125
- Klein G, Andronick J, Elphinstone K, Heiser G, Cock D, Derrin P, Elkaduwe D, Engelhardt K (2010) seL4: formal verification of an operating system kernel. *Commun ACM* 53(6):107–115
- Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R, Heiser G (2014) Comprehensive formal verification of an os microkernel. *ACM Trans Comput Syst* 32(1):2:1–2:70
- Klein G, Andronick J, Kuz I, Murray T, Heiser G, Fernandez M (2018) Formally verified software in the real world. *Commun ACM* 61(10):68–77
- Koenig J, Shao Z (2018) Refinement-based game semantics for certified abstraction layers. In: *Proceedings of the 35th annual ACM/IEEE symposium on logic in computer science*, Saarbrücken, Germany, pp 633–647
- Leroy X (2009) Formal verification of a realistic compiler. *Commun ACM* 52(7):107–115
- Liang H, Feng X (2016) A program logic for concurrent objects under fair scheduling. In: *Proceedings of 43th ACM symposium on principles of programming languages*, St. Petersburg, FL, USA, pp 385–399
- Liang H, Feng X (2018) Progress of concurrent objects with partial methods. In: *Proceedings of the 45th ACM symposium on principles of programming languages*, Los Angeles, CA, USA, pp 20:1–20:31
- Liu M, Rieg L, Shao Z, Gu R, Costanzo D, Kim J, Yoon M (2020) Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. In: *Proceedings of the 47th ACM symposium on principles of programming languages*, New Orleans, LA, USA, pp 20:1–20:31
- Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: A proof assistant for higher-order logic. Springer Verlag, Berlin
- Shapiro J, Doerrie MS, Northup E, Miller M (2004) Towards a verified, general-purpose operating system kernel. In: *Proceedings of the 1st NICTA workshop on operating system verification*, Sydney, Australia, pp 1–19
- Shin J, Kim J, Honore W, Vanzetto H, Radhakrishnan S, Balakrishnan M, Shao Z (2019) Wormspace: a modular foundation for simple, verifiable distributed systems. In: *Proceedings of the ACM symposium on cloud computing*, Santa Cruz, California, pp 299–311
- Stampoulis A (2012) VeriML: a dependently-typed, user-extensible, and language-centric approach to proof assistant. PhD thesis, Yale University
- Walker BJ, Kemmerer RA, Popek GJ (1980) Specification and verification of the UCLA Unix security kernel. *Commun ACM* 23(2):118–131
- Wang Y, Wilke P, Shao Z (2019) An abstract stack based approach to verified compositional compilation to machine code. In: *Proceedings of the ACM on Programming Languages*, Lisbon, Portugal, pp 62:1–62:30
- Xu F, Fu M, Feng X, Zhang X, Zhang H, Li Z (2016) A practical verification framework for preemptive os kernels. In: *Proceedings of the 28th international conference on computer aided verification*, Toronto, Canada, Toronto, Canada, pp 59–79

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.