VTOS: Research on Methodology of "Light-Weight" Formal Design and Verification for Microkernel OS

Zhenjiang Qian[1,2,3,⋆], Hao Huang[1,2], and Fangmin Song[1,2]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210046, China
[2] Department of Computer Science and Technology, Nanjing University, Nanjing 210046, China
[3] Department of Informatics, King's College London, London WC2R 2LS, United Kingdom
zhenjiang.qian@gmail.com, {hhuang,fmsong}@nju.edu.cn

Abstract. The correctness of the operating systems is difficult to be described with the quantitative methods, because of the complexity. Using the rigorous formal methods to verify the correctness of the operating systems is a recognized method. The existing projects of formal design and verification focus on the validation of code level. In this paper, we present a "light-weight" formal method of design and verification for OS. We propose an OS state automaton model (OSSA) as a link between the system design and verification, and describe the correctness specifications of the system based on this model. We implement the trusted operating system (verified trusted operating system, VTOS) as a prototype, to illustrate the method of consistency verification of system design and safety requirements with formalized theorem prover Isabelle/HOL. The result shows that this approach is feasible.

Keywords: Microkernel OS, Formal Design, Formal Verification, System Correctness.

1 Introduction

Operating system (OS), as a significant system software or platform, provides services and security protection for a variety of other applications. The correctness of OS is the core issue of information security, and how to elaborate or ensure the correctness of OS is the direction of industry and academia efforts. Because of the enormous size and complexity of OS, the accuracy is not easy

⋆ This work is supported by the National High Technology Research and Development Program (863 Program) of China under grant No. 2011AA01A202, the National Science Foundation of China under grant No. 61021062, the "Six Talents Peak" High-Level Personnel Project of Jiangsu Province under grant No. 2011-DZXX-035, University Natural Science Research Program of Jiangsu Province under grant No. 12KJB520001.

S. Qing et al. (Eds.): ICICS 2013, LNCS 8233, pp. 17–32, 2013.

to be described and illustrated. Despite intensive testing, the bugs in OS do occur over time, which can be seen from the fact that the current mainstream commercial OSs continually release the update patches.

For low-assurance application environment, using test to ensure the correctness of underlying OS can be considered sufficient. The situation is quite different for high-assurance application environment, in which even as complete as possible coverage of test cases cannot guarantee that OS is correctly implemented.

An apparently better approach is to use a more rigorous mathematical approach to do formal description and verification, e.g. code analysis, model checking [1] and theorem proving. Formal methods can guarantee the correctness of the software program. But in the actual application process, the developers often shun it due to the division of abstraction levels, and the complexity and scale of the program, as well as the difference between programming and formal logic. Therefore, many scholars try the "light-weight" [2-4] tools to do formal description and design, which have powerful expressiveness ability and can be applied easily. At present, many scholars do verification in the code level for the implemented OS. The OS codes (usually written with C language) are transformed into the input syntax for interactive mechanized verification using the theorem prove tools, e.g. Isabelle [5] or Coq [6]. There are two problems for the work: the investment of time and persons is great, such as seL4 [7] and Verisoft XT [31] projects, and for system maintenance and upgrade work, the updated OS codes still need to be transformed and verified with the existing OS modules merged.

In this paper, we argue that in order to design and implement the OS with formal methods, and illustrate the correctness of the system and ensure the system's security, we need to verify whether the design of OS meets the requirements of system security, and thereby verify whether the implementation meets the requirements of design. We propose that not only the verification for system implementation (code level) needs the formal methods, but also the system design (design level) requires the use of formal logic to ensure the correctness of the design, to the greatest extent of the correctness of system.

In this paper, we present a "light-weight" formal method for the design and verification of OS. We propose an OS state automaton model (OSSA) as a link between the system design and verification, and describe the correctness conditions of the system based on this model. We implement the trusted operating system (verified trusted operating system, VTOS) as a prototype, to illustrate the method of consistency verification of the system design and safety requirements with formalized theorem prover Isabelle/HOL.

The rest of this paper is organized as follows. Section 2 reviews the related work of formal design and verification for OS. Section 3 describes the state automaton model (OSSA). Section 4 illustrates the method of formal verification for VTOS in Isabelle/HOL. Section 5 explains the concrete verification of VTOS in Isabelle/HOL. Section 6 concludes this paper and makes prospect for the future work.

2 Related Work

In 1978, UCLA developed UCLA Secure UNIX [8] for PDP-11 machine. In this system, developers gave multi-layer specification. Top-level specification described permission access control model of the kernel. The specification of the abstraction layer included abstract data structures. The low-level specification contained all the variables and objects used in the kernel call interface. The lowest level is the Pascal codes of the kernel. The authors verified the consistency of specifications inside several parts of the abstraction levels, but did not prove the consistency between all the kernel levels, and consistency of implementation.

Provably Secure Operating System (PSOS) was developed by SRI international during 1973-1980, aiming at providing a generic OS with provable security. PSOS proposed multi-level hierarchical abstraction, and used a specification language (SPECIAL) [9] for precisely defining modules of all levels as well as abstraction mapping between layers. PSOS provided only some simple examples to illustrate the consistency of its implementation and specification, and did not process the formal design and verification from strict sense.

In 1995, Charlie Landau led the EROS [10] project, which focused its formal verification mainly on the correctness of the address translation and the security of usage of kernel memory. The Coyotos project, led by Hopkins in 2006, as the successor of the EROS, proposed a low-level programming language (BitC) [11] and defined the corresponding formal semantics.

VFiasco project [12] organized by the Technical University of Dresden, verified the microkernel compatible with L4. VFiasco used the SPIN model checker to verify the IPC mechanism, and used the PVS [13] verification system as an assistive theorem prover to build system model and verify the correctness of the code. VFiasco modified the C++ language to enhance security of the programming language. In 2008, they reported the Nova Hypervisor project [14] as the continuation of VFiasco, proposed formal models for IA32 processor and memory, and implemented a tool directly converts C++ code to corresponding PVS semantic code.

The seL4 project [7] was initiated by the Laboratory of National ICT Australia (NICTA) during 2004-2006. The project mainly focused on verifying L4 microkernel of the ARM architecture with Isabelle/HOL prover [5]. In the report [15] Klein illustrated the validation work. The objective of formalization was to verify that the implementation was consistent with the expected definition of the abstract model. They are dedicated to the reliability [16] and integrity [17] of OS with the formal methods and improving the efficiency of the real systems [18].

From beginning of the 1990s to the present, the Flint team led by Professor Shao at Yale University has done significant work on formal verification [19][20]. In safe language, they designed a new programming language (VeriML) [21], which introduces the inductive definition of data types in the logic system λHOL^{ind} based on λHOL logic [22], and provides strong formal description capacity as well as features of type security. The open logical framework OCAP [23] developed by Feng et al successfully combined the validation logics of

different modules in OS, to form a complete verification system, and to ensure the scalability of the verification model. Meanwhile, the Flint team also studies the methodology of verifying the concurrent management [24] and uses the hierarchical abstraction method to validate the various functional modules in the OS [25]. The Flint team cooperates with the group led by Professor Chen in University of Science and Technology of China. They are committed to program verification technology in high trusted software, and studies how to effectively integrate the form of program verification, proof-carry-code (PCC) and domain-specific language, in order to form new methods for improving productivity of writing robust software, correctness and security [26-30].

Verisoft is a project of large computer system design from 2003 to 2007, to provide formal verifications for the entire computer system from hardware to application layer, i.e., from bottom up (Pervasive Formal Verification) [31][32]. In 2007, its successor Verisoft XT project was officially launched. In Verisoft XT project, pervasive formalization means that the entire project focuses on not only the compiler, or the correctness of the model of machine instructions, but that the design of the entire system must undergo the rigorous formal verification, and form a complete verification chain of software and hardware [33-36].

3 State Automaton Model of VTOS (OSSA)

In this section, we illustrate the functionality of our self-implemented secure microkernel OS (VTOS), and analyse the elements of the state automaton model, i.e., software/hardware computing entities, system object states, and events. Based on these elements, we explain the state automaton model of VTOS (OSSA).

3.1 Architecture and Functionality of VTOS

VTOS is our self-implemented secure and trusted OS. The kernel of VTOS provides the most general services, e.g., process/thread schedule, interrupt handling, message service and simple I/O service, etc. Other functionality services including file management (FM), process management (PM), and memory management (MM), etc, are implemented as user mode processes. The architecture VTOS is shown in the Figure 1.

VTOS adopts the microkernel architecture, in which the inter-process communication (IPC) is realized with message mechanism. From the aspect of functionality, the message processing copies the messages in the message buffer of sender to the one of receiver. The message processing needs to check the legality of the target process, and look up message buffers. Meanwhile, the microkernel converts the hardware and software interrupts into messages. The hardware interrupts are generated by hardware, and the software interrupts are the only way for the user-level applications to request system services delivered to the kernel.

The microkernel handles the process scheduling, responsible for state transitions of the processes in ready, running, and blocking.

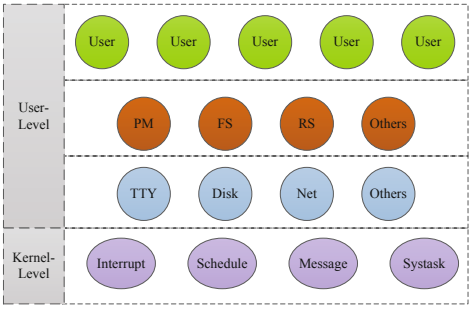

Fig. 1. The architecture of VTOS

Due to isolation, the processes outside the microkernel are prohibited to perform the actual I/O operations, manipulate kernel tables or complete the functionality as in monolithic-kernel OS. The microkernel works in the sealed condition, and modules outside the microkernel execute as independent processes and their codes are kept unchanged. Because of these limitations, the microkernel should provide a set of services for system drivers and functionality servers. These special services are handled by the system task, which is scheduled as a separate process, and is responsible for receiving all of request messages from system drivers and functionality servers, and forwarding these messages to the microkernel.

For interrupt handling, microkernel stores the process context, e.g. information of registers, and do procedure of interrupt handling, and restore the interrupted processes.

The principal reason for us to choose the microkernel architecture is that the microkernel is the only execution object running in highest privileged level and it is isolated. The device drivers are all implemented as user mode programs. The new OS service modules, if needed, are added in user mode. Therefore the user mode programs can not damage the microkernel directly. And the microkernel OS is adequate for multi-cores or many-cores platforms [37].

3.2 Hardware and Software Computing Entities

Modern computer system consists of memory, arithmetic units, and control units, etc. The memory includes registers in the CPUs, caches, main memory, registers in the device controllers, and exchange area in hard disks. The required instruction sequence and data for each arithmetic unit, and the results calculated are all stored in the memory. There is little trivial difference between these concepts and normal ones in computer organization, and these concepts are more convenient for introducing the following abstract model of OS.

The objects that can affect the data in the memory include arithmetic units in the CPU cores, and device controllers with DMA mechanism. All of these objects read and write the data in memory in parallel, and we call these objects as ***hardware computing entities***.

The control units in the CPU control the running of arithmetic unit according to the values of the data in the memory. For example, in the Intel CPU, if CPL bit in the $EFLAG$ register is not equal to 0, the arithmetic unit may not execute the $SETGDT$ instruction. Because data in the memory (as $EFLAGS$ in Intel CPU) contains the mask bits of interrupt, it also affects the selection of interrupt events to be handled in the next step. The number of clock cycles for different instructions vary. At the end of each clock cycle, some instructions may happen to complete exactly, while others need to continue in the next clock cycle. Meanwhile, at the end of each clock cycle, there may be a number of events to arrive. According to the current value of each memory location and the hardware computing entities that have just finished an instruction, the control units select a number of events to be handled in these hardware computing entities from the beginning of the next clock cycle. Therefore, the value of each memory location is the key part of the system state. We call the object that consists of a sequence of instructions and can independently run in a hardware computing entity, as a ***software computing entity***, e.g., processes, threads, and function objects.

We suppose U is a software computing entity. The read-only data object set of U is denoted by $R(U)$. And the modified data object set of U is denoted by $W(U)$. We call the object set $R(U) \cup W(U)$ as a ***working object set*** of U, which is denote by $RW(U)$. Because of interrupt mechanism, the software computing entity U may be interrupted during execution, and it may wait in the waiting queue. During the waiting period of U, any other software computing entity do not access its working object set. We can divide all the current software computing entities in the system into two categories. One is the set of ***running software computing entities*** denoted by A_r, in which all the entities are running on respective hardware computing entities. The other is the set of ***waiting software computing entities*** denoted by A_w, in which all the entities do not possess the hardware computing entities, thus in the waiting queue. The set of all the software computing entities is denoted as A, i.e., $A = A_r \cup A_w$.

3.3 System Object State

In this subsection, we use the notation in [38] to describe the system state.

Suppose $A_r = \{a_1, a_2, \cdots, a_t\}$, $A_w = \{a_{t+1}, a_{t+2}, \cdots, a_n\}$, and the working object set of a_i is $RW(a_i) = \{x_{ij} \mid j = 1, 2, \cdots, n_i;\ i = 1, 2, \cdots, n\}$. We denote the value range of object x_{ij} by V_{ij}. The initial value of the object x_{ij} is s^0_{ij}. We suppose that a_i has m_i instructions, and after running k instructions, the value of the object x_{ij} in $RW(a_i)$ will be $s^k_{ij}, k = 1, 2, \cdots, m_i$. The value space of the working object set $RW(a_i)$ is

$$\prod_{j=1}^{n_i} V_{ij} \tag{1}$$

which we denote as $VRW(a_i)$. The semantics of the software computing entity a_i may be expressed as the mapping: $\prod_{j=1}^{n_i} V_{ij} \to \prod_{j=1}^{n_i} V_{ij}$.

We call the Cartesian product

$$\prod_{i=1}^{n}\prod_{j=1}^{n_i} V_{ij} \tag{2}$$

as the ***system object state set*** denoted by S_D.

Sometimes, the software computing entity a_i may create or release objects during its execution process. In this case, its working object set $RW(a_i)$ may vary and so is its value space $\prod_{j=1}^{n_i} V_{ij}$.

In the system memory, in addition to the objects in the equation (2), there are a lot of fragments of free storage space. The positions and number of these fragments vary constantly. In order to analyze the system object state, we make unified numbering for all the data storage units including register, cache, memory, etc. These numbers as whole is a subset of the natural number set $\mathcal{N}$, and we denote this subset as N, $N \subset \mathcal{N}$.

3.4 Events

During each clock cycle, one device may generate hardware interrupt, that possible changes the set A_r. The process may generate the software interrupt and change A_r. And the exceptions of CPU may also change set A_r. We call the hard interrupt, software interrupt, and exceptions generated during the clock cycle as ***current arrival events*** denoted by E_a. The events that arrive before or are waiting to be handled are called as ***waiting events*** denoted by E_w. And the events being handled are called as ***running events*** denoted by E_r. Obviously, the system object state S_D, the current arrival events E_a, the waiting events E_w, the running software computing entities A_r, and the waiting software computing entities A_w, determine the next E_r, E_w, A_r and A_w.

3.5 State Model of VTOS

Base on the above analysis, the running of modern computer system can be described as the progress in which several computing entities manipulate a series of data objects in the memory in parallel.

From the point of view of the software, the factors affecting the system running are the data and instruction sequences in the memory and the arrival events. So the computer system can be described as a state automaton (OSSA) as follows.

1. The set of the accepted events corresponds to the alphabet of OSSA.
2. The events handled during the period from power on to system halt correspond to a sentence accepted by OSSA.
3. At the end of a clock cycle, the values of each memory location and the arrival events correspond to the current state of OSSA.
4. The modification actions to the memory units by hardware computing entities correspond to the state transitions of OSSA.

5. The state in which the CPU executes the *halt* instruction corresponds to the termination state of OSSA.

We build the OSSA model of VTOS as follows.

Definition 1 (VTOS OSSA Model)**.** *VTOS OSSA model is a state automaton,*

$$A_{OS} = (S, \Sigma, \delta, s_0, \Gamma) \tag{3}$$

and the definitions of $S, \Sigma, \delta, s_0, \Gamma$ *are as follows:*

1. The system state: $S = (S_D, A_r, A_w, E_r, E_w)$
   (a) The object state S_D: the system object state.
   (b) The running software computing entities A_r: the software computing entities that are running in hardware computing entities;
   (c) The waiting software computing entities A_w: the software computing entities that are waiting for running;
   (d) The running events E_r: the events that are chosen to be handled;
   (e) The waiting events E_w: the events that wait to be handled;
2. Σ is the set of all kind of events that the OS accepts.
3. The set of state transfer functions denoted $\delta : \delta(s, E_a) = s'$.
4. s_0 is the initial state of VTOS.
5. Γ is the termination state set, $\Gamma \subseteq S$. Whenever the system reaches a state in Γ, the system terminates.

The architecture of VTOS OSSA model is shown as in Figure 2.

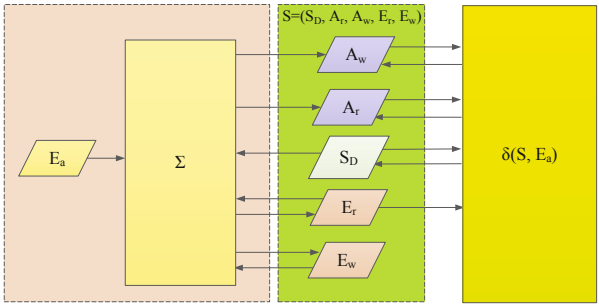

Fig. 2. OSSA model of VTOS

4 Method of Formal Verification for VTOS in Isabelle/HOL

In section 3, we explain how to use the OSSA model to describe the design of VTOS. Based on the OSSA model, in this section, we illustrate the method of how to use Isabelle/HOL theorem proving tools to verify the consistency of the design and implementation of VTOS.

4.1 Introduction of Isabelle/HOL Theorem Proving

Isabelle [5] is a theorem prover tool for validation of the abstract problems described by the logic systems. Isabelle can rigorously verify the program logics in the computer system. Isabelle/HOL supports for Higher-Order Logic in Isabelle, and provide the interactive verification platform with the form of functional programming.

Isabelle/HOL is a type system, and has predefined series of basic types, e.g., *nat*, *int* and *list*. Users can also define new types by keywords *record* and *datatype*, data objects by keyword $definition$, functions by keywords *primrec* and fun, and formulae by *lemma* and *theorem*. The special domain is established by building the theory in Isabelle/HOL. In general, each theory is a collection of types, definitions, functions, theorems and proofs.

Isabelle/HOL provides many rules for proving theorems. Meanwhile, we can quote existing theories that contain one or more proven theorems.

4.2 Domain of VTOS in Isabelle/HOL

As described in section 3 above, it is obvious that the system object state can be represented as the elements of the special domain[38][39], that is a mathematical system. The system states, the operations of the hardware and software computing entities, and the properties of the system shown in the form of propositions constitute the domain of VTOS, denoted by $\mathbf{M}_{Computer}$. Relatively, the domain of VTOS in Isabelle/HOL is denoted by $\mathbf{M}_{Isabelle/HOL}$. According to the design and implementation of VTOS, we construct the domain $\mathbf{M}_{Isabelle/HOL}$. There is the isomorphism[39] relation between $\mathbf{M}_{Computer}$ and $\mathbf{M}_{Isabelle/HOL}$, i.e., the proposition in $\mathbf{M}_{computer}$ is true if and only if the corresponding proposition is true in the $\mathbf{M}_{Isabelle/HOL}$.

According to the above described, we establish the relationship between VTOS and Isabelle/HOL logic reasoning system. First, we construct the OSSA model, and based on this model, we design and implement VTOS. With the implementation of VTOS, we describe the domains $\mathbf{M}_{computer}$, and $\mathbf{M}_{Isabelle/HOL}$ in Isabelle/HOL that is isomorphic to the $\mathbf{M}_{computer}$. Thereafter, the properties about the functionality or security of VTOS can be mapped to logic formulae in $\mathbf{M}_{Isabelle/HOL}$. On this basis, we verify the consistency of design and implementation of VTOS through reasoning in the Isabelle/HOL logic system.

4.3 OSSA Model of VTOS in Isabelle/HOL

We regard OS as a server that provides its services whenever a user-level program claims the requests. After finishing the service, the OS gets ready to receive another request. As described in subsection 3.5, an OS works like an OSSA $A_{OSSA} = (S, \Sigma, \delta, s_0, \Gamma)$. The principal elements of OSSA are the system state set S and the set of state transfer functions δ. Each element in S is a vector that consists of values of the data objects of all the current processes. All the functions of event handling and functionality constitute the set of state transfer

functions δ. What each function in δ does is that it transfer the initial values of the data objects to new ones according to the functionality semantics.

For describing the OS domain, there are two principal works, i.e., design of the working object set, and of the functions of event handling and functionality that operate on the working object set. The processing of functions in OS corresponds to the transitions of values of the working object set. Therefore, in order to show that VTOS accomplishes all its functions correctly, we prove that VTOS correctly transfer the initial values of the data objects according to the functionality semantics whenever it handles the events and service requests. Relatively, in order to illustrate the security of VTOS, we prove that the system states satisfy the security specifications at any time.

Now we explain how to describe VTOS as an OSSA. As mentioned in subsection 3.1, VTOS is a microkernel OS for general purpose, consisting of microkernel, file manage management, process management, memory management, and device drivers. Firstly, we design the working object set $\{x_{ij} \mid j = 1, 2, \cdots, n_i\}$ for each computing object M_i, i.e., the software computing entity. Suppose the object x_{ij} takes value on the set $\{V_{ij} \mid j = 1, 2, \cdots, n_i\}$, the possible states for M_i is the space $\prod_{j=1}^{n_i} V_{ij}$. Then we design the function set $\{f_{ij} \mid j = 1, 2, \cdots, m_i\}$ for M_i. This function set is the subset of state transfer functions δ. The semantics of the function f_{ij} can be expressed as the mapping from S to S. In order to illustrate the correctness of the function f_{ij}, we prove that the running of the function f_{ij} complies with the expected functionality semantics.

4.4 State Transition Functions of OSSA in Isabelle/HOL

From the aspect of view of composition, the functions implemented in VTOS has two aspects, i.e., the working objects and the instruction sequences. In order to illustrate the correctness of functions, we describe the working objects and the instruction sequences in Isabelle/HOL. As shown in the following definitions of *Instr*, *PCinc* and *NextS*, we define the type of instructions as *Instr*, that contains all kind of assembly instructions, e.g., *mov*, *add*, *push*, *pop*, *leave*, etc. Due to the space limitations, we introduce some of them. The case *movrr Register Register*, whose type is *Instr*, corresponds to the instruction "*movl Register Register*". The function *PCinc* adds 1 to the program counter register *PC*. The semantics of the function *NextS* is that for "$s' = NextS\ s\ instr$", after executing the instruction *instr*, the system state s is converted into s'. Similarly, by applying *NextS* to the sequences of the functionality instructions, we can calculate the state transitions for multi-steps.

Definitions of Instr, PCinc and NextS in Isabelle/HOL

```
datatype Instr =
           movrr  Register Register
         | movir  int   Register
         | addrr  Register Register
         ...
```

```
fun PCinc :: "state => state"
where
"PCinc s = s (| R:=((R s) (pc:=((R s) pc)+1)) |)"

fun NextS :: "state => Instr => state"
where
"NextS s (movrr y x) = PCinc (s (| R:=((R s) (x:=((R s) y)))|))"
 ...
```

4.5 Proving the Integrity Property in Isabelle/HOL

The running OS is a collection of several processes. Each process provides designated services. Can these processes always provide the services during the running of system? How can we describe and judge that these processes provide the services correctly? These questions confuse many researchers. As illustrated above, we regard that VTOS corresponds to an OSSA model. We prove the immutability of the corresponding OSSA model of VTOS to demonstrate the integrity of VTOS.

As illustrated in section 3, the current values of all working objects of the processes in the OS correspond to the system state of OSSA. It is obvious that if the accepted alphabet and transition functions of the OSSA remain unchanged, the OSSA is immutable. In order to prove this characteristic conveniently, during the design of VTOS, we guarantee that the working objects of any two processes do not intersect, and that the possible target address set of the branch points in the functions is identified and will not be changed by user applications and inputs. Because the services provided by VTOS are identified, this objective is reasonable. With these criteria We design and implement VTOS, and the result shows that it is feasible. Therefore we need only to prove that the accepted events and the semantics of functions in the microkernel are kept unchanged, i.e., that all memory units occupied by the codes of microkernel remain unchanged, and that the selection of target address in each branch point is consistent with the semantics of the functionality. This actually proves that all the factors that may affect the integrity of VTOS are kept unchanged during the running of system.

5 Verification of VTOS in Isabelle/HOL

In this section, we take the module of message processing service as the example to describe the method of proving the correctness of event handling and state transitions.

For building the model or structure in Isabelle/HOL, the significant work is to construct the domain and define the mapping or interpretation from Isabelle/HOL to the domain.

While we design the working object set for the modules of the VTOS, we have designed the state set of OSSA actually. And while we design the system call functions, we have designed the state transition functions of OSSA actually,

and while we establish the integrity conditions, we have constructed the relations on domain actually. So when we complete VTOS design, we also complete the construction of the domain.

As illustrated in subsection 3.4, whenever the events occur, the OS select the corresponding event handlers to execute. The events may be interrupts, and service requests, etc. It is important to prove the correctness of the execution of event handlers and the corresponding state transitions.

In the microkernel of VTOS, the module of message processing service mainly includes the functions of *sys_send*, and *sys_receive*, etc. The working objects of these functions are components of the PCBs (process control block) of the sender and receiver processes.

In the following sections, we illustrate the correctness proof of the function *sys_send* in VTOS, which sends message from one process to another process. The main part of the C codes for *sys_send*, the corresponding assembly codes and definition in Isabelle/HOL are shown as follows.

sys_send in C

```
int sys_send(struct proc *caller_ptr, int dst, message *m_ptr,
             unsigned flags)
{
    struct proc *dst_ptr = get_proc_from_pid(dst);
    ...
    copy_mess(m_ptr, dst_ptr->p_messbuf, sizeof(message));
    ...
}
```

sys_send in ASM

```
<sys_send>:
push    %ebp
movl    %esp,%ebp
subl    $0x28,%esp
movl    0xc(%ebp),%eax
movl    %eax,(%esp)
call    c0101128
...
movl    %eax,0x4(%esp)
movl    0x10(%ebp),%eax
movl    %eax,(%esp)
call    c010116d <copy_mess>
...
```

sys_send in Isabelle/HOL

```
definition sys_send :: "Code"
where
  "sys_send =
    pushr        ebp;
    movrr        esp ebp;
    subir        10 esp;
    movirr       3 ebp eax;
    movrir       eax 0 esp;
    call   get_proc_from_pid;
    ...
    movrir       eax 1 esp;
    movirr       4 ebp eax;
    movrir       eax 0 esp;
    call     copy_mess;
    ..."
```

It is the key point for us to establish the formulae to describe the specifications for the correctness of the event handlers and the state transition functions whenever the event handlers are called, because these conditions should fit to

any starting state for the state transition functions. For example, we illustrate the correctness specification for *sys_send* as follows.

$$\forall s.\ Q(s) \wedge (s' = NextnS\ s\ sys\_send) \longrightarrow P(s, s') \tag{4}$$

in which "$NextnS\ s\ sys\_send$" means the state after execution of the functionality semantics of *sys_send*, i.e., the state for multi-steps. The formula 4 means when the starting state s satisfies the condition Q, the function *sys_send* can correctly send designated messages to target process, i.e., that the starting state s and the subsequent state s' satisfy condition P. We regard that not all the states are security states. Therefore, there are some states that the VTOS may not reach, and we need not consider these cases for the starting states. In the formula 4, the condition P is considered as the semantic formula of the function *sys_send*, and Q as the initial condition.

From the aspect of view of the working object set, we consider the starting state condition Q. The working object set of the function *sys_send* includes the actual parameters in the stack, and the accessibility of the corresponding components of the PCBs of the sender and receiver processes. So the condition Q is defined as follows:

$$\begin{aligned} Q(s) : (s.regs.sp + 1 = caller\_ptr) \wedge \\ (s.regs.sp + 2 = dst) \wedge \\ (s.regs.sp + 3 = m\_ptr) \end{aligned} \tag{5}$$

The formula 5 states that the actual parameters that the function *sys_send* needs are located in the stack at proper location and possess the correct values.

When the sender process requires to send the message to the receiver process, the target process may be waiting for this message, or in dealing with other messages. Here we explain the case that the target process is waiting for this message. In this case, the handler *sys_send* copies the designated length of bytes from the sender's message buffer to the receiver's. Therefore, the correctness condition for *sys_send* is that the values in the two relevant memory location interval, i.e., the message buffers of the sender and receiver, are equivalent. The semantic formula P is defined as follows.

$$\begin{aligned} P(s, s') = CmpM(s', s.M(s.regs.sp + 3), \\ s'.M(proc + sizeof\_proc * \\ s.M(s.regs.sp + 2) + p\_messbuf), \\ sizeof\_msg) \end{aligned} \tag{6}$$

in which the auxiliary formula $CmpM$ is defined as follows.

$$\begin{aligned} CmpM(s, p, q, n) = (i \geq 0 \wedge i \leq (n - 1)) \\ \longrightarrow\ (s.M(p + i) = s.M(q + i)) \end{aligned} \tag{7}$$

The amount of the verification codes in Isabelle/HOL for the whole VTOS is about 56K SLOC. The result shows that VTOS achieves the desired safety.

6 Conclusion and Future Work

In this paper, we present a "lightweight" formal method to design and implement a safe and reliable OS. We propose a state automaton (OSSA) model as the basis of the system design. With this model, we describe the system states and state transition functions. We use Isabelle/HOL theorem proving tool to establish its corresponding formal model, and define the specifications of the system, to prove that the system design and implementation comply with these specifications. With this method we achieve a safe and trusted operating system (VTOS). The results show that this approach is feasible.

Various functional modules in the OS are often designed using a variety of program logics, and involved in different levels of abstraction, such as C language, assembly codes, and hardware layers, etc. For the correctness of the whole system integrated with these functional modules, it is not simply the conjunction of the correctness of each module. For the future work, we will study how to combine the verification of the separated modules to illustrate the correctness of the whole system, from the aspect of view of the domain and type theory.

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
2. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11(2), 256–290 (2002)
3. Denney, R.: Succeeding with Use Cases: Working Smart to Deliver Quality. Addison-Wesley Professional Publishing, Boston (2005)
4. Agerholm, S., Larsen, P.G.: A lightweight approach to formal methods. In: Hutter, D., Stephan, W., Traverso, P., Ullmann, M. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 168–183. Springer, Heidelberg (1999)
5. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
6. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
7. Klein, G., Andronick, J., Elphinstone, K., et al.: seL4: Formal verification of an operating system kernel. Communications of the ACM 53(6), 107–115 (2010)
8. Walker, B.J., Kemmerer, R.A., Popek, G.J.: Specification and verification of the UCLA Unix security kernel. Communications of the ACM 23(2), 118–131 (1980)
9. Robinson, L., Roubine, O.: Special: A Specification and Assertion Language. Technical Report, Stanford Research Institute (1977)
10. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: A fast capability system. In: 17th SOSP, pp. 170–185. ACM, New York (1999)
11. Shapiro, J.S., Sridhar, S., Doerrie, M.S.: BitC Language Specification. Technical Report (1996)
12. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: the VFiasco project. Technical Report (2002)
13. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)

14. Tews, H., Weber, T., Volp, M., Poll, E., Eekelen, M., Rossum, P.: Nova Micro-Hypervisor Verification Formal, machine-checked verification of one module of the kernel source code. Technical Report (2008)
15. Klein, G., Elphinstone, K., Heiser, G., et al.: seL4: Formal verification of an OS kernel. In: 22nd SOSP, pp. 207–220. ACM, New York (2009)
16. Heiser, G., Murray, T., Klein, G.: It's time for trustworthy systems. In: 33rd S & P, pp. 67–70. IEEE Computer Society, Washington (2012)
17. Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 enforces integrity. In: Van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 325–340. Springer, Heidelberg (2011)
18. Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A.: Timing analysis of a protected operating system kernel. In: 32nd RTSS, pp. 339–348. IEEE Computer Society, Washington (2011)
19. Shao, Z.: Certified Software. Communications of the ACM 53(12), 56–66 (2010)
20. Stampoulis, A., Shao, Z.: Static and User-Extensible Proof Checking. In: 39th POPL, pp. 273–284. ACM, New York (2012)
21. Stampoulis, A., Shao, Z.: VeriML: Typed Computation of Logical Terms inside a Language with Effects. In: 15th ICFP, pp. 333–344. ACM, New York (2010)
22. Barendregt, H.P., Geuvers, H.: Proof-assistants using dependent type systems. Elsevier, Amsterdam (1999)
23. Feng, X.: An Open Framework for Certified System Software. Ph.D. dissertation. Yale University, New Haven (2007)
24. Guo, Y., Feng, X., Shao, Z., Shi, P.: Modular Verification of Concurrent Thread Management. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 315–331. Springer, Heidelberg (2012)
25. Vaynberg, A., Shao, Z.: Compositional Verification of a Baby Virtual Memory Manager. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 143–159. Springer, Heidelberg (2012)
26. Liang, H.J., Feng, X., Fu, M.: A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations. In: 39th POPL, pp. 455–468. ACM, New York (2012)
27. Tan, G., Shao, Z., Feng, X., Cai, H.X.: Weak Updates and Separation Logic. New Generation Comput. 29(1), 3–29 (2011)
28. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about Optimistic Concurrency Using a Program Logic for History. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 388–402. Springer, Heidelberg (2010)
29. Ferreira, R., Feng, X., Shao, Z.: Parameterized Memory Models and Concurrent Separation Logic. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 267–286. Springer, Heidelberg (2010)
30. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: 30th PLDI, pp. 170–182. ACM, New York (2008)
31. Alkassar, E., Hillebrand, M.A., Leinenbach, D., Schirmer, N.W., Starostin, A.: The Verisoft Approach to Systems Verification. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 209–224. Springer, Heidelberg (2008)
32. Daum, M., Dorrenbacher, J., Bogan, S.: Model stack for the pervasive verification of a microkernel-based operating system. In: 5th VERIFY, pp. 56–70. CEUR-WS.org, Aachen (2008)
33. Alkassar, E., Cohen, E., Hillebrand, M.A., Kovalev, M., Paul, W.J.: Verifying shadow page table algorithms. In: 10th FMCAD, pp. 267–270. IEEE Press, New York (2010)

34. Alkassar, E., Cohen, E., Hillebrand, M.A., Pentchev, H.: Modular specification and verification of interprocess communication. In: 10th FMCAD, pp. 167–174. IEEE Press, New York (2010)
35. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Ingredients of operating system correctness. In: Embedded World 2010 Conference (2010)
36. Baumann, C., Bormer, T., Blasum, H., Tverdyshev, S.: Proving memory separation in a microkernel by code level verification. In: 14th ISORCW, pp. 25–32. IEEE Computer Society, Washington (2011)
37. Wentzlaff, D., Agarwal, A.: Factored Operating Systems (FOS): The Case for a Scalable Operating System for Multicores. ACM SIGOPS Operating Systems Review 43(2), 76–85 (2009)
38. Li, W.: Mathematical Logic: Basic Principles and Formal Calculus. Science China Press, Beijing (2007) (in Chinese)
39. Marker, D.: Model Theory An Introduction. Oxford University Press, Oxford (1990)