操作系统形式化设计与安全需求的一致性验证研究

钱振江[1),2),3)] 黄 皓[1)] 宋方敏[1)]

[1)](南京大学计算机科学与技术系 南京 210046)

[2)](常熟理工学院计算机科学与工程学院 江苏 常熟 215500)

[3)](伦敦大学国王学院 伦敦 英国 WC2R 2LS)

摘 要 采用数学形式化方法对操作系统进行设计和验证可以保证系统的高度安全性.目前已有的操作系统形式化研究工作主要是验证系统的实现在代码级的程序正确性.提出一种操作系统形式化设计和验证的方法,采用操作系统对象语义模型(OSOSM)对系统的设计进行形式化建模,使用带有时序逻辑的高阶逻辑对操作系统的安全需求进行分析和定义.对象语义模型作为系统设计和形式化验证的联系.以实现和验证过的可信微内核操作系统VTOS为实例,阐述形式化设计和安全需求分析,并使用定理证明器Isabelle/HOL①对系统的设计和安全需求的一致性进行验证,表明VTOS达到预期的安全性.

关键词 操作系统;形式化设计;安全需求;一致性验证;定理证明;信息安全;网络安全

中图法分类号 TP316 **DOI号** 10.3724/SP.J.1016.2014.01082

Research on Consistency Verification of Formal Design and Security Requirements for Operating System

QIAN Zhen-Jiang[1),2),3)] HUANG Hao[1)] SONG Fang-Min[1)]

[1)](*Department of Computer Science and Technology, Nanjing University, Nanjing* 210046)

[2)](*School of Computer Science and Engineering, Changshu Institute of Technology, Changshu, Jiangsu* 215500)

[3)](*King's College London, London WC2R 2LS, UK*)

Abstract The mathematical formal methods for operating system design and verification achieve high assurance of system security. The existing formalization research works in the scope of operating system mainly focus on showing that an implementation complies with the program correctness in the code-level verification. In this paper, we propose a method for formal design and verification. We adopt the operating system object semantics model (OSOSM) for formal modeling of the system design, and analyze and define the security requirements using higher-order logic (HOL) with temporal logic (TL). We view OSOSM as the link between system design and formal verification, and take the self-implemented and verified trusted operating system (VTOS) as an example to illustrate formal design and analysis of security requirements. Meanwhile, we use the theorem prover Isabelle/HOL to verify the consistency between system design and security requirements, and show that VTOS achieves the desired security.

Keywords operating system; formal design; security requirements; consistency verification; theorem proving; information security; network security

收稿日期:2012-04-25;**最终修改稿收到日期**:2013-12-30.**本课题得到国家自然科学基金创新研究群体基金**(60721002)、**江苏省"六大人才高峰"高层次人才项目**(2011-DZXX-035)、**江苏省高校自然科学研究项目**(12KJB520001)**和** CSLG **科研项目**(QT1312)**资助**.**钱振江**,**男**,1982**年生**,**博士**,**讲师**,**中国计算机学会**(CCF)**会员**,**主要研究方向为操作系统安全与形式化验证、嵌入式系统**. E-mail: zhenjiang.qian@gmail.com.**黄 皓**,**男**,1957**年生**,**博士**,**教授**,**博士生导师**,**主要研究领域为系统软件、信息安全**.**宋方敏**,**男**,1961**年生**,**博士**,**教授**,**博士生导师**,**主要研究领域为数理逻辑和量子计算机**.

① Isabelle/HOL website. http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html

1 引　言

操作系统(Operating System，OS)作为系统软件，由于其复杂性，其设计和实现的正确性很难用定量的方式进行描述和说明，而操作系统的正确性是信息安全的基础.采用形式化的方法对操作系统进行设计和验证[1-3]是操作系统领域公认的标准方法.澳大利亚 NICTA 实验室主导的 seL4 操作系统项目[1,4]在系统设计过程中采用函数式语言 Haskell[5]来搭建系统原型，方便后期直接转换为形式化定理证明器 Isabelle/HOL[6]所需的程序逻辑输入. seL4 对系统按照不同的实现层次进行抽象，通过说明各种抽象层次在功能语义上的一致性来达到验证的目的[7-8]. Verisoft 项目提出对计算机系统进行普适形式化验证(pervasive formal verification)[2]，以实现对操作系统从底层硬件抽象到高层应用软件的完全验证[9]，验证的层次是代码级的验证，即系统实现的正确性验证[10].耶鲁大学 Shao 等人领导的 Flint 项目对系统的程序验证逻辑进行改进[3]，主要是对系统的各种功能模块的混合验证方法进行改进，目的是提高验证的效率[11-12].

本文认为，采用形式化方法对操作系统进行设计和实现，以此来说明系统的正确性和保证系统的安全性，为达到这样的目的，首先需要使用形式逻辑验证操作系统的设计是否满足对系统的安全需求，进而验证操作系统的实现是否满足设计的要求.本文提出，不仅对系统实现(代码级)的验证需要使用形式化的方法，在系统设计(设计级)的过程中就需要使用形式逻辑来保证设计的正确性，从而最大程度上保证系统的正确性.

本文提出一种形式化设计和验证相结合的方法，使用对象语义模型(Operating System Object Semantics Model，OSOSM)[13]作为系统设计和验证之间的联系，利用包含时序逻辑(Temporal Logic，TL)[14]的高阶逻辑(Higher-Order Logic，HOL)[15]描述系统的安全需求，并以我们实现的可信操作系统(Verified Trusted Operating System，VTOS)为例，阐述使用形式化定理证明器 Isabelle/HOL 对系统的设计和安全需求的一致性进行验证的方法.

本文第 2 节以 VTOS 微内核为例，采用微内核 OS 对象语义模型进行形式化建模；第 3 节对微内核 OS 的安全需求进行分析，并使用带有时序逻辑的高阶逻辑进行严格定义；第 4 节阐述使用定理证明器 Isabelle/HOL 对形式化设计与安全需求的一致性验证；第 5 节对本文的工作进行总结，并对我们进一步的研究方向进行展望.

2 对象语义模型

本节阐述对象语义模型 OSOSM 的基本框架. OSOSM 是一种针对操作系统的框架语义模型[13]，将系统中各种行为的执行功效看成是对系统状态的改变或者迁移，对应地，将系统行为的主客体作为对象来看待，系统状态的转化看成是系统中行为的主客体对象相互作用的结果. OSOSM 对系统的描述采用状态机的方式，使用状态轨迹来表达系统的运行变化.

2.1 OS 对象的抽象

OS 为其他各种功能性的上层应用软件提供平台的支撑，是一个服务系统，对其他功能模块提出的服务请求、以及中断事件作出响应. OS 对服务请求和中断事件的响应是通过系统行为来完成的，系统行为的主体通过对内核或者用户数据客体的检索、读取、修改或者创建等动作来完成功效.例如用户进程通过发送消息给服务进程请求进行系统调用，OS 的消息处理行为根据用户进程和服务器的进程控制块、内核数据、系统状态等对象来完成功效.我们可以将系统行为的主客体抽象为对象，系统行为的功效是通过主客体的相互作用来完成的，进而实现对系统状态的改变.

在这样的对象抽象的场景下，下一小节重点阐述 OSOSM 的整体框架.

2.2 OSOSM 框架

OSOSM 采用分层结构，是一种开放式的框架语义模型，可以描述 OS 的功能模块的行为语义. OSOSM 按照基本功效、具体实现、实现优化的递进层次来分层，区别于按照软件功能模块和硬件抽象的分层方式.为此，OSOSM 层级结构包含基本功效层、实现层和优化层.基本功效层描述系统行为动作对系统状态改变的基本功效，关注系统状态改变的结果，不涉及处理的具体细节.实现层描述为了实现基本功效层的功能功效而引入的其他关键对象和行为语义函数.与指称语义[16]的概念相同，OSOSM 使用对象集合上的语义函数来描述系统行为的功能语义.优化层描述为了提高实现层的语义函数和具体实现细节的效率和性能而引入的新的对象和语义函数.

为了表述的方便，我们以完全自主研发的安全可信操作系统 VTOS 为例来阐述 OSOSM 框架. VTOS 是一种微内核架构的 OS，设计和实现部分均经过形式化的验证，达到一定的安全标准级别. VTOS 拥有安全核(secure kernel)和访问控制机制，实现了虚拟内存管理、多核和多线程管理、文件系统管理以及安全网关等功能. 限于篇幅，我们结合 VTOS 微内核部分的形式化设计和验证来阐述.

VTOS 微内核为服务进程以及设备驱动程序等提供功能服务，主要负责消息处理、进程调度以及中断处理.

VTOS 微内核 OSOSM 框架如图 1 所示. 以下小节，对 VTOS 微内核的功能语义进行分解，包括对象以及行为语义.

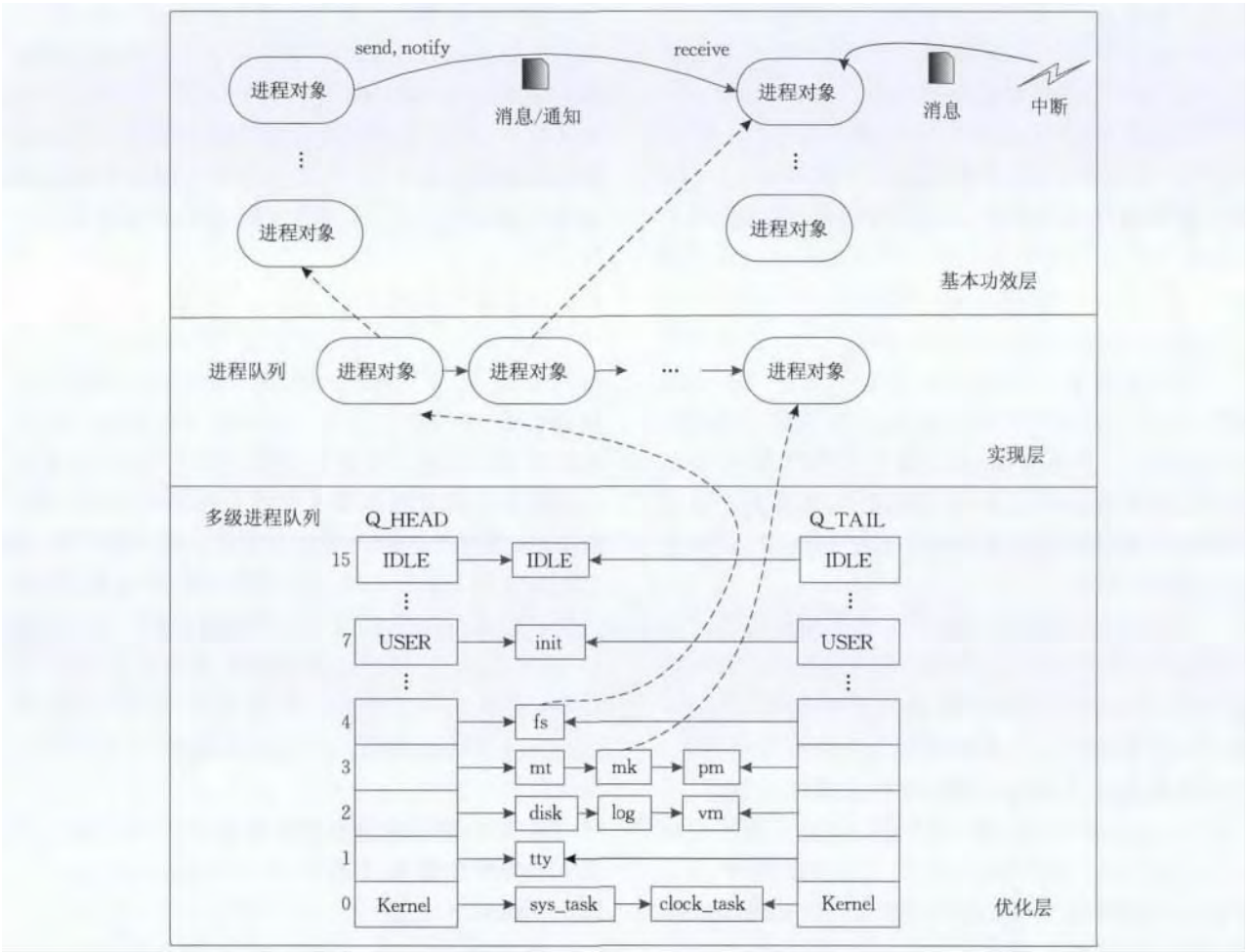

图 1 VTOS 微内核 OSOSM 框架

2.2.1 基本功效层

(1) 对象集合

基本功效层对象集合 M_1 包含以下对象：

① 消息 $message=(m\_source, m\_type, m\_content)$，$m\_source$ 为消息发送进程对象标识，m_type 为消息类型，$m\_content$ 为消息体；

② 进程对象 $process=(proc\_nr, p\_messbuf, p\_rts\_flags, p\_getfrom, p\_sendto)$，$proc\_nr$ 为进程对象标识，$p\_messbuf$ 为消息缓冲区，p_rts_flags 指示进程对象是否处于等待发送消息状态或者是等待接收状态，取值如 $RECEIVING$ 和 $SENDING$；$p\_getfrom$ 为希望接收消息的目标进程对象标识；p_sendto 为希望发送消息的目标进程对象标识；

③ 系统状态 $state=(pset, process\_q, sys\_buffer, intr\_source, next\_running\_proc)$，$pset$ 为当前进程对象的集合，$process\_q$ 为就绪进程队列；sys_buffer 为系统缓冲区，$intr\_source$ 为中断源；$next\_running\_proc$ 为下一个将要运行的进程对象标识；

④ 系统状态集 S，S 为系统状态 $state$ 的集合.

(2) 行为语义

① 消息处理行为.

在基本功效层，从功效上来说，微内核的消息处理行为是将发送者进程对象所指示的消息缓冲区中的消息体复制到接收者进程对象所指示的消息缓冲

区中. 在这一层，我们主要考虑系统行为功效的完成.

假设在状态 s 时进程 $p1$ 向进程 $p2$ 发送消息，在状态 w 时完成这一功效. 其功效用逻辑公式表示如下：

$$
\begin{aligned}
&send\ p1\ p2:\\
&w.pset.p2.p\_messbuf :=\\
&\begin{cases} s.pset.p1.p\_messbuf \\ \quad \text{if } s.pset.p2.p\_rts\_flags = RECEIVING \wedge \\ \qquad (s.pset.p2.p\_getfrom = ANY \vee \\ \qquad s.pset.p2.p\_getfrom = p1) \\ \text{invariable if other} \end{cases}
\end{aligned}
$$

假设在状态 s 时进程 $p2$ 接收进程对象 $p1$ 的消息，在状态 w 时完成这一功效. 其功效用逻辑公式表示如下：

$$
\begin{aligned}
&receive\ p2\ p1:\\
&w.pset.p2.p\_messbuf :=\\
&\begin{cases} s.pset.p1.p\_messbuf \\ \quad \text{if } s.pset.p1.p\_rts\_flags = SENDING \wedge \\ \qquad s.pset.p1.p\_sendto = p2 \\ \text{invariable if other} \end{cases}
\end{aligned}
$$

② 进程调度行为.

VTOS 微内核的进程调度行为包括 *schedule*，*dequeue* 和 *pick_proc* 操作行为.

schedule 是将被调度的进程对象加入到就绪进程队列中. 假设在 s 状态下调度进程对象 p，在状态 w 时完成这一功效. 其功效用逻辑公式表示如下：

$$
\begin{aligned}
&schedule\ p:\\
&w.process\_q := s.process\_q \cup \{p\}
\end{aligned}
$$

dequeue 将进程对象从进程队列中去除. 如果一个进程对象被阻塞，那么这个进程对象将从进程队列中去除. 假设在 s 状态下阻塞进程对象 p，在状态 w 时完成这一功效. 其功效用逻辑公式表示如下：

$$
\begin{aligned}
&dequeue\ p:\\
&w.process\_q := s.process\_q \setminus \{p\}
\end{aligned}
$$

pick_proc 选择下一个将要运行的进程. 在基本功效层，*pick_proc* 不进行调度优化，为此选择进程队列首部的进程对象占有处理器 CPU 资源，这主要通过对系统状态 *state* 中 *next_running_proc* 域进行赋值来完成. VTOS 微内核将在下一个时钟中断处理完成后切换 *next_running_proc* 域指向的进程对象来运行. 关于调度优化部分，将在优化层进行阐述. 假设在 s 状态下选择进程对象 p 占有 CPU 资源，在状态 w 时完成这一功效. 其功效用逻辑公式表示如下：

$$
\begin{aligned}
&pick\_proc:\\
&w.next\_running\_proc := \text{head}\ s.process\_q
\end{aligned}
$$

其中 head 操作子表示取队列的首部.

③ 中断处理行为.

VTOS 微内核对硬件中断的处理，主要是通过发送消息给驱动程序进程对象，由驱动程序进程对象来进行相应的中断服务处理. 在 VTOS 微内核对象语义模型中，从对象层次来说，VTOS 微内核的中断处理行为的功效是根据不同的中断源，对系统缓冲区对象进行设置. 本文主要描述键盘中断和磁盘中断. 假设在 s 状态下发生键盘中断或者磁盘中断，在状态 w 时完成这一功效. 其功效用逻辑公式表示如下：

$$
\begin{aligned}
&interrupt\_handle:\\
&w.sys\_buffer :=\\
&\begin{cases} data\_from\_disk \ \text{if } s.intr\_source = INTR\_DISK \\ data\_from\_input \\ \qquad \text{if } s.intr\_source = INTR\_KEYBOARD \\ \text{invariable if other} \end{cases}
\end{aligned}
$$

其中 *data_from_disk* 表示从磁盘输入的数据，*data_from_input* 表示从键盘输入的数据.

2.2.2　实现层

实现层阐述实现基本功效层的系统行为功效所涉及的其他数据对象和相关的语义函数.

我们在设计上，对 VTOS 消息处理行为采用“汇合”机制，对方进程对象不在接收消息的情况下，发送进程将会阻塞；类似地，对方进程对象不在发送消息的情况下，接收进程将会阻塞. 同时，由于各种执行主体作为独立的进程对象运行，消息处理的“汇合”机制导致系统中进程对象的过分同步，造成性能的下降，有可能引起死锁现象. 为此，在实现层，我们在 VTOS 的消息处理行为中同时加入异步机制，引入一种特殊的消息方式，称为“通知”. 如果接收者不在等待消息或者通知，通知的发送者也不会阻塞. 通知信息不会丢失，该通知只是简单地被挂起. 当目标进程执行接收动作时，通知将优先于消息被处理. 在通知的设计上，通知体只包含发送者的标识信息，不包含具体的内容信息，接收者如果需要其他的必要信息，可以通过向发送者发送消息来请求相应的内容.

(1) 对象集合

实现层对象集合 M_2 的定义如下：

① M_2 包含基本功效层对象集合 M_1；

② 进程对象 *process*. 实现层的进程对象 *process* 在基本功效层的基础上增加相应的数据成员对象：*p_lastcall*，*s_notify_pending* 和 *s_msg_pending*. *p_lastcall* 为进程对象最近一次消息行为语义的标识信息. 由于消息处理行为的语义函数需要保存结果信息，为此我们在进程 *process* 对象中引入 *p_lastcall* 对象来保存语义函数信息. *p_lastcall*=(*nr_sys_call*, *result*)，其中 *nr_sys_call* 是行为标识，取值如 *SEND*, *RECEIVE* 和 *NOTIFY*；*result* 是语义函数结果信息，取值如 *ELOCKED*(标识死锁)，*OK*(标识消息处理行为完成，发送或者接收行为成功或者进程对象阻塞)；*s_notify_pending* 是记录所有向该进程对象发送通知的进程对象标识链表；*s_msg_pending* 是记录所有向该进程对象发送消息的进程对象标识链表.

(2) 行为语义

实现层行为语义主要包括消息处理行为 *send*, *notify* 和 *receive*.

① 消息的处理行为 *send*.

假设在状态 s 时进程对象 $p1$ 向进程对象 $p2$ 发送消息，在状态 w 时完成这一功效. *send* 行为在实现层的逻辑语义分情况描述如下：

i. 当目标进程对象 $p2$ 也在发送消息，并且该消息的目标进程是 $p1$，那么就会产生死锁. 此时，系统状态 *state* 的变化主要为：进程对象 $p1$ 中的 *p_lastcall* 值变为(*SEND*, *ELOCKED*)，即记录发送进程对象在进行发送消息行为时发生死锁，等待系统的处理，其语义表示如下：

$w.pset.p1.p\_lastcall := (SEND, ELOCKED)$

ii. 当目标进程 $p2$ 正在等待消息，并且等待消息的来源进程标识是 *ANY* 或是发送进程 $p1$，那么消息就成功发送，并修改目标进程 $p2$ 的状态信息，将目标进程 $p2$ 唤醒加入到进程队列，其语义表示如下：

$w.pset.p1.p\_lastcall := (SEND, OK)$
$w.pset.p2.p\_messbuf := s.pset.p1.p\_messbuf$
$w.pset.p2.p\_rts\_flags :=$
$\quad s.pset.p2.p\_rts\_flags - RECEIVING$
$schedule(p2)$

iii. 当上面两种情况都不符合的情况下，发送进程 $p1$ 进入阻塞状态，同时修改发送进程对象信息，并将发送进程对象 $p1$ 加入到进程对象 $p2$ 的等待发送消息进程对象链表(*s_msg_pending* 域)，其语义表示如下：

$w.pset.p1.p\_lastcall := (SEND, OK)$
$w.pset.p1.p\_rts\_flags :=$
$\quad s.pset.p1.p\_rts\_flags + SENDING$
$w.pset.p1.p\_sendto := p2$
$w.pset.p2.s\_msg\_pending :=$
$\quad s.pset.p2.s\_msg\_pending \# p1$
$dequeue(p1)$

② 通知的发送行为 *notify*.

假设在状态 s 时进程对象 $p1$ 向进程对象 $p2$ 发送通知，在状态 w 时完成这一功效. *notify* 行为在实现层的逻辑语义分情况描述如下：

i. 如果进程对象 $p2$ 处于接收状态，即 $w.pset.p2.p\_rts\_flags = RECEIVING$，并且在等待 $p1$ 或者 *ANY*，那么 $p1$ 直接将通知发送给 $p2$，并修改 $p2$ 的状态，将 $p2$ 唤醒加入到进程队列，其语义表示如下：

$w.pset.p1.p\_lastcall := (NOTIFY, OK)$
$w.pset.p2.p\_messbuf := (p1, Notify, NULL)$
$w.pset.p2.p\_rts\_flags :=$
$\quad s.pset.p2.p\_rts\_flags - RECEIVING$
$schedule(p2)$

ii. 如果进程对象 $p2$ 不处于接收状态，通知不能成功发送，发送进程不会被阻塞. 同时，该通知需要被挂起，以便将来进程对象 $p2$ 能进行处理，我们采用的方法是在进程对象 $p2$ 的 *s_notify_pending* 链表中记录下 $p1$ 的标识信息. 其语义表示如下：

$w.pset.p1.p\_lastcall := (NOTIFY, OK)$
$w.pset.p2.s\_notify\_pending :=$
$\quad s.pset.p2.s\_notify\_pending \# p1$

③ 消息的接收行为 *receive*.

假设在状态 s 时进程对象 $p2$ 对发送给它的通知和消息进行处理，在状态 w 时完成这一功效. *receive* 行为在实现层的逻辑语义分情况描述如下：

i. 如果存在进程对象正在向进程对象 $p2$ 发送通知，则优先处理，这主要通过查看进程对象 $p2$ 的 *s_notify_pending* 域来确定. *s_notify_pending* 记录的通知信息从链表首部开始处理. 由于采用异步模式发送通知，为此不需要唤醒链表首部对应的发送进程. 其语义表示如下：

$w.pset.p2.p\_lastcall := (RECEIVE, OK)$
$w.pset.p2.p\_messbuf :=$
$\quad (\text{head } s.pset.p2.s\_notify\_pending, Notify, NULL)$

$w.pset.p2.s\_notify\_pending :=$
　tail $s.pset.p2.s\_notify\_pending$

其中，tail 操作子表示取链表除首部以外的部分；

ii. 如果不存在向进程对象 $p2$ 发送的通知，但存在向进程对象 $p2$ 发送的消息，这通过 $p2.s\_msg\_pending$ 域进行判断. 对于消息的处理，如果进程对象 $p2$ 的接收对象（$p\_getfrom$ 域）为 ANY，则顺序从链表首部开始处理；否则，检查接收对象是否在 $s\_msg\_pending$ 链表对象中，并进行相应的处理. 其语义表示如下：

$w.pset.p2.p\_lastcall := (RECEIVE, OK)$
$w.pset.p2.p\_messbuf :=$
$\begin{cases} w.pset.(\text{head } s.pset.p2.s\_msg\_pending).p\_messbuf \\ \quad \text{if } s.pset.p2.p\_getfrom = ANY \\ w.pset.(s.pset.p2.p\_getfrom).p\_messbuf \\ \quad \text{if } s.pset.p2.p\_getfrom \neq ANY \wedge \\ \quad s.pset.p2.p\_getfrom \in s.pset.p2.s\_msg\_pending \\ \text{invariable if other} \end{cases}$
$w.pset.(\text{head } s.pset.p2.s\_msg\_pending).p\_rts\_flags :=$
　$w.pset.(\text{head } s.pset.p2.s\_msg\_pending).p\_rts\_flags - RECEIVING$
　　if $s.pset.p2.p\_getfrom = ANY$
$w.pset.(s.pset.p2.p\_getfrom).p\_rts\_flags :=$
　$s.pset.(s.pset.p2.p\_getfrom).p\_rts\_flags - RECEIVING$
　　if $s.pset.p2.p\_getfrom \neq ANY \wedge$
　　$s.pset.p2.p\_getfrom \in s.pset.p2.s\_msg\_pending$
$schedule(\text{head } s.pset.p2.s\_msg\_pending)$
　　if $s.pset.p2.p\_getfrom = ANY$
$schedule(s.pset.p2.p\_getfrom)$
　　if $s.pset.p2.p\_getfrom \neq ANY \wedge$
　　$s.pset.p2.p\_getfrom \in s.pset.p2.s\_msg\_pending$
$w.pset.p2.s\_msg\_pending :=$
$\begin{cases} \text{tail } s.pset.p2.s\_msg\_pending \\ \quad \text{if } s.pset.p2.p\_getfrom = ANY \\ s.pset.p2.s\_msg\_pending \backslash s.pset.p2.p\_getfrom \\ \quad \text{if } s.pset.p2.p\_getfrom \neq ANY \wedge \\ \quad s.pset.p2.p\_getfrom \in s.pset.p2.s\_msg\_pending \\ \text{invariable if other} \end{cases}$

iii. 如果进程对象 $p2$ 的接收对象不处于发送状态，那么接收进程 $p2$ 将被阻塞. 其语义表示如下：

　　$w.pset.p2.p\_lastcall := (RECEIVE, OK)$
　　$w.pset.p2.p\_rts\_flags :=$
　　　$s.pset.p2.p\_rts\_flags + RECEIVING$
　　$dequeue(p2)$

2.2.3　优化层

优化层描述为了提高实现层的语义函数和具体实现细节的效率和性能而引入的新的对象和语义函数. 为了高效地组织进程队列，我们按照进程优先级引入多级进程队列.

(1) 对象集合

优化层对象集合 M_3 定义如下：

① M_3 包含实现层对象集合 M_2；

② 进程对象 $process$. 优化层的进程对象 $process$ 在基本功效层和实现层的基础上增加相应的数据成员对象：$p\_priority$ 和 $p\_ticks\_left.p\_priority$ 是进程对象的优先级，VTOS 在设计上包含 16 级优先级队列，取值范围为 0～15，0 为最高级；p_ticks_left 是进程对象的剩余时间片，用于衡量和计算进程对象的优先级；

③ 系统状态 $state$，优化层的系统状态 $state$ 在基本功效层和实现层的基础上增加相应的数据成员对象：$prev\_proc$，记录当前状态的上一个运行的进程对象；

④ 在基本功效层和实现层中，系统状态 $state$ 的 $process\_q$ 数据域是一个进程队列对象，由于我们在优化层引入多级进程队列，为此 $process\_q$ 数据域在优化层作为多级进程队列来看待，具体为一个从进程优先级到对应优先级的进程队列的映射，即输入一个进程优先级，得到该优先级对应的进程队列，从而实现多优先级进程队列.

(2) 行为语义

针对优化层引入的多级进程队列对象，系统行为语义主要涉及对多级进程队列对象的操作，包括进程调度行为 $schedule$，$dequeue$ 和 $pick\_proc$.

多级进程队列入队操作行为 $schedule$. 在基本功效层，$schedule$ 行为将就绪进程对象插入到进程队列中. 在优化层，由于引入了多优先级进程队列，为此首先需要根据进程的优先级和时间计算进程应该位于哪个队列，以及放在队列的首部或者尾部位置.

假设在状态 s 时，系统执行“将进程对象 p 放进多优先级进程队列”的动作，在状态 w 时完成这一功效. $schedule$ 在优化层的逻辑语义分情况描述如下：

① 当进程对象 p 的 p_ticks_left 域不为 0 时，表示进程对象 p 时间片未用完，将进程 p 放入进程对象优先级 $p\_priority$ 域所对应的优先级队列的首部，其语义表示如下：

$w.process\_q\ \ s.pset.p.p\_priority :=$
$p \sharp (s.process\_q\ \ s.pset.p.p\_priority)$
if $s.pset.p.p\_ticks\_left \neq 0$

② 当进程对象 p 的 p_ticks_left 域为 0，并且进程对象 p 不是内核进程（$proc\_nr > 0$），同时在系统的上一时间片已占有 CPU 资源（$w.prev\_proc = p$），并且进程对象 p 不是最低优先级（$p\_priority < 15$）时，那么应该降低进程对象 p 的优先级，VTOS 采用优先级增加 1 来实现，同时将进程对象 p 放入新的优先级队列的尾部，其语义表示如下：

$w.pset.p.p\_priority := s.pset.p.p\_priority + 1$
$w.process\_q\ (s.pset.p.p\_priority + 1) :=$
$(s.process\_q\ (s.pset.p.p\_priority + 1)) \sharp p$
if $s.pset.p.p\_ticks\_left = 0 \wedge$
$s.pset.p.proc\_nr > 0 \wedge$
$s.prev\_proc = p \wedge$
$s.pset.p.p\_priority < 15$

③ 当进程对象 p 的 p_ticks_left 域为 0，并且进程对象 p 不是内核进程（$proc\_nr > 0$），同时进程对象 p 在系统的上一时间片未占有 CPU 资源（$w.prev\_proc \neq p$），并且进程对象 p 不是最高优先级（$p\_priority > 0$）时，那么应该提高进程对象 p 的优先级，VTOS 采用优先级减少 1 来实现，同时将进程对象 p 放入新的优先级队列的尾部，其语义表示如下：

$w.pset.p.p\_priority := s.pset.p.p\_priority - 1$
$w.process\_q\ (s.pset.p.p\_priority - 1) :=$
$(s.process\_q\ (s.pset.p.p\_priority - 1)) \sharp p$
if $s.pset.p.p\_ticks\_left = 0 \wedge$
$s.pset.p.proc\_nr > 0 \wedge$
$s.prev\_proc \neq p \wedge$
$s.pset.p.p\_priority > 0$

④ 对于其他情况，将进程对象 p 放入进程对象优先级 $p\_priority$ 域所对应的优先级队列的尾部，其语义表示如下：

$w.process\_q\ \ s.pset.p.p\_priority :=$
$(s.process\_q\ \ s.pset.p.p\_priority) \sharp p$ if other

多级进程队列出队操作行为 *dequeue*.

对应地，*dequeue* 行为将进程对象从多优先级队列中去除，其语义表示如下：

$w.process\_q\ \ s.pset.p.p\_priority :=$
$(s.process\_q\ \ s.pset.p.p\_priority) \backslash p$

进程选择操作行为 *pick_proc*. *pick_proc* 行为在基本功效层选择下一个将要运行的进程，由于在优化层引入多优先级进程队列，*pick_proc* 将在多级队列中选择就绪进程，按照优先级从高到低的顺序检查各级队列，每个队列中从首部开始选取. 假设在 s 状态下选择多级进程队列中的进程对象占有 CPU 资源，在状态 w 时完成这一功效. 其语义表示如下：

pick_proc：
$w.next\_running\_proc := pick\_p\ s\ 0\ 15$

子过程 *pick_p t begin end* 定义如下：

$$pick\_p\ t\ begin\ end :=$$
$$\begin{cases} pick\_p\ t\ (begin+1)\ end \\ \quad \text{if } t.process\_q\ begin = \text{NULL} \wedge begin < end \\ \text{head}\ (t.process\_q\ begin) \\ \quad \text{if } t.process\_q\ begin \neq \text{NULL} \\ IDLE\ \ \text{if other} \end{cases}$$

3 VTOS 系统安全需求

本节阐述 VTOS 系统的安全需求分析，并使用带有时序逻辑的高阶逻辑对安全需求进行了严格的定义.

对于 VTOS 微内核部分，考虑到安全需求的多样性以及可验证性，我们主要分析的安全需求类型包括完整性、隔离性和机密性. 对于完整性，传统的理解认为在系统运行过程中需要保证系统代码的完整性和数据完整性，即代码和数据不可被恶意地修改，这些可以通过访问控制策略实现. 除了代码和数据完整性，我们认为还需要做到系统功效的完整性，即系统的功能行为始终能完成系统设计所期望的效果. 对于隔离性，主要包括进程空间隔离性和进程行为的隔离性. 进程空间隔离性是指进程对象的页表相互之间不存在交集. 进程行为隔离性是指进程在运行过程中，相互之间不会干扰对方的行为. 对于机密性，进程对象无法访问其他进程对象的私有数据部分，即进程对象的私有数据不会泄露给其他进程对象.

限于篇幅，本文主要从系统功效完整性和进程行为隔离性这两方面，来对 VTOS 微内核的安全需求进行分析、定义和验证.

OSOSM 以高阶逻辑（Higher-Order Logic，HOL）为元逻辑，支持基本的逻辑命题演算，并且包括简单类型理论（simple type theory）和类型化的 λ 演算（typed lambda calculus），提供丰富的对象表达能力. 在第 2 节的基础上，我们使用 VTOS 微内核的语义函数并结合时序逻辑来阐述 VTOS 的安全性

目标，其中时序逻辑主要是对安全需求中与时间有关的问题进行描述.

下一小节首先对需求描述用到的时序逻辑进行说明.

3.1　时序逻辑

时序逻辑用于描述系统关于时间问题的需求，包括线性时序逻辑(Linear Temporal Logic，LTL)和计算树逻辑(Computation Tree Logic，CTL). LTL 对时间的描述是线性(严格偏序)的，即在时间序列上任意两个时间点都是有前后关系的；CTL 主要用于描述分支时间(branching-time)序列结构，即在时间序列上每个时间点允许存在分支的不同直接前驱和直接后继时间节点.

时序逻辑中讨论的是时间序列，而 OSOSM 采用状态机的方式对系统进行描述，即描述的是状态序列. 为此我们在使用时序逻辑对 VTOS 的安全需求进行描述时，将状态序列作为时间序列来看待. 这存在一个问题，时间序列不存在“回归”现象，是一种树结构，而状态序列可能存在“回归”现象，即环结构，如 $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_0$，这与时间序列存在不一致. 为此我们对状态序列的描述进行不回归处理，如上述的状态序列表示成 $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$，其中 $s_0 = s_5$.

VTOS 在每个状态下可能的后继状态是不确定的，这主要由系统的行为动作决定，因此是分支状态序列结构. 我们采用 CTL 对 VTOS 的安全需求进行描述.

CTL 的时序描述符主要包括：路径分支描述 A/E 和序列路径描述 G/F/X. 假设 θ 为分支路径命题，时序描述符 A/E 定义如下：

$s \models A\theta$ 表示“从状态 s 开始，对于所有状态路径，命题 θ 成立”；

$s \models E\theta$ 表示“从状态 s 开始，存在某条状态路径，该路径上命题 θ 成立”；

假设 φ 为序列路径命题，即对于单条路径的命题，时序描述符 G/F/X 定义如下：

$s \models F\varphi$ 表示“从状态 s 开始，在单路径上将来某个状态，命题 φ 成立”；

$s \models G\varphi$ 表示“从状态 s 开始，在单路径上将来所有状态，命题 φ 成立”；

$s \models X\varphi$ 表示“在单路径上，对于状态 s 的下一个状态，命题 φ 成立”.

路径分支和序列路径描述符相结合如图 2 所示.

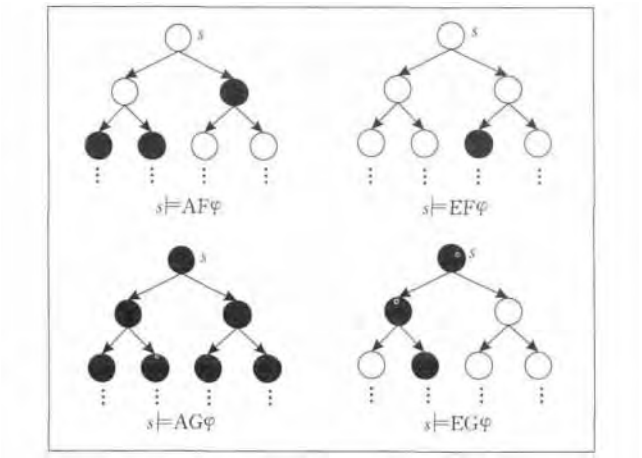

图 2　时序逻辑

CTL 中没有表示“动作对状态路径分支的影响”的方法，我们对其进行扩展，使用“$s, action \vdash \beta$”来表示“在当前状态 s 下执行 $action$ 动作后，在后续状态路径会满足 β”，即当前状态 s 的直接后继根据 $action$ 动作可以得到确定，如图 3 所示.

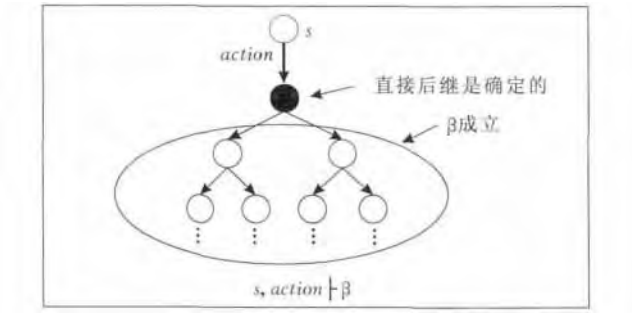

图 3　动作对状态路径分支的影响

以下章节从系统功效完整性和进程行为隔离性这两方面，并从消息处理、进程调度和中断处理的角度，开始对 VTOS 微内核行为的安全需求进行分析.

3.2　消息处理行为的功效完整性安全需求

(1) 消息发送行为 *send* 能正确地将发送者进程对象所指示的消息缓冲区中的消息体复制到接收者进程对象所指示的消息缓冲区中. 谓词公式表示如下：

$$\forall s \in S, p1 :: process, p2 :: process.(s, (send\ p1\ p2) \vdash EF(\lambda w :: state.w.pset.p2.p\_messbuf = s.pset.p1.p\_messbuf)) \quad (\varphi 1)$$

其中，λ 运算用于对完成功效的状态进行抽象.

对应地，通知发送行为 *notify* 能正确地将通知发送到接收者进程对象. 谓词公式表示如下：

$$\forall s \in S, p1 :: process, p2 :: process.(s, (notify\ p1\ p2) \vdash EF(\lambda w :: state.w.pset.p2.p\_messbuf = (p1, Notify, \mathrm{NULL}))) \quad (\varphi 2)$$

对于接收行为 *receive*，由于消息、通知发送行为的正确性都依赖于接收行为，在 $\varphi 1$ 和 $\varphi 2$ 安全需求中已隐含接收行为的功效完整性，为此不单独对接收行为 *receive* 的功效完整性进行定义.

(2) 假设系统初始状态 $s0$ 为可信状态，在 $s0$ 的将来任何时刻，微内核的消息处理行为都能完成上述第 1 点的功能需求. 谓词公式表示如下：

$$s0 \models \mathrm{AG}(\varphi 1 \wedge \varphi 2) \qquad (\varphi 3)$$

3.3 进程调度行为的功效完整性安全需求

(1) 微内核的多级进程队列入队操作行为 *schedule* 能正确地将进程加入到多级进程队列中. 谓词公式表示如下：

$$\forall s \in S,\ p :: process.(s,(schedule\ p)) \vdash \mathrm{AX}\ (\lambda w :: state.\ \exists i.0 \le i \le 15 \wedge p \in (w.process\_q\ i))) \qquad (\varphi 4)$$

微内核的多级进程队列出队操作行为 *dequeue* 能正确地将进程从多级进程队列中删除. 谓词公式表示如下：

$$\forall s \in S, p :: process.(s,(dequeue\ p)) \vdash \mathrm{AX}\ (\lambda w :: state.\ \forall i.0 \le i \le 15 \rightarrow p \notin (w.process\_q\ i))) \qquad (\varphi 5)$$

微内核的进程选择操作行为 *pick_proc* 能正确地将 CPU 资源分配给多级进程队列中的进程对象. 谓词公式表示如下：

$$\forall s \in S.(s, pick\_proc \vdash \mathrm{AX}(\lambda w :: state.w.next\_running\_proc = pick\_p\ s\ 0\ 15)) \qquad (\varphi 6)$$

(2) 假设系统初始状态 $s0$ 为可信状态，在 $s0$ 的将来任何时刻，微内核的进程调度行为都能完成上述第 1 点的功能需求. 谓词公式表示如下：

$$s0 \models \mathrm{AG}(\varphi 4 \wedge \varphi 5 \wedge \varphi 6) \qquad (\varphi 7)$$

3.4 中断处理行为的功效完整性安全需求

(1) 微内核的中断处理行为能正确地根据不同的中断源，对系统缓冲区对象进行设置. 谓词公式表示如下：

$$\forall s \in S.(s, interrupt\_handle \vdash \mathrm{AX}\ (\lambda w :: state.\ (s.intr\_source = INTR\_DISK \rightarrow w.sys\_buffer = data\_from\_disk) \vee (s.intr\_source = INTR\_KEYBOARD \rightarrow w.sys\_buffer = data\_from\_input))) \qquad (\varphi 8)$$

(2) 假设系统初始状态 $s0$ 为可信状态，在 $s0$ 的将来任何时刻，微内核的中断处理行为都能完成上述第 1 点的功能需求. 谓词公式表示如下：

$$s0 \models \mathrm{AG}(\varphi 8) \qquad (\varphi 9)$$

3.5 进程行为隔离性需求

微内核的进程行为不会受到其他进程对象的干扰，换句话说微内核的上述消息处理、进程调度以及中断处理行为在初始状态 $s0$ 为可信的情况下，任何后续状态都能完成期望的功效，谓词公式表示如下：

$$s0 \models \mathrm{AG}\ (\varphi 1 \wedge \varphi 2 \wedge \varphi 4 \wedge \varphi 5 \wedge \varphi 6 \wedge \varphi 8) \qquad (\varphi 10)$$

上述安全需求以初始状态 $s0$ 是可信状态为前提，我们通过可信启动来保证这一点. VTOS 启动过程的安全保护由基于 TPM 的可信启动来完成. 我们在 VTOS 中整合了 tboot 软件，通过对 VTOS 启动过程中系统镜像进行验证来防止对系统的恶意篡改和破坏，保证系统运行的初始状态的可信性.

4 VTOS 设计与安全需求的一致性验证

本节阐述 VTOS 形式化设计与安全需求一致性的验证. 在第 2 节，我们使用 OSOSM 描述 VTOS 的设计，为此，一致性验证即是验证 OSOSM 模型的语义是否符合第 3 节提出的安全需求.

我们借助定理证明器 Isabelle/HOL 来实现整个验证过程. 验证方法分成 3 个部分：(1) 利用 Isabelle/HOL 对第 2 节描述的 OSOSM 模型进行形式化建模；(2) 安全需求的时序逻辑部分在 Isabelle/HOL 中的验证方法；(3) 系统设计与安全需求的一致性验证.

4.1 Isabelle/HOL 验证系统

这一小节介绍验证过程将用到的 Isabelle/HOL 验证系统. Isabelle 是一种定理证明器，主要用于验证使用逻辑系统描述的抽象问题，可以对计算机系统的程序逻辑进行严格的验证. Isabelle/HOL 是对高阶逻辑的支持，采用函数式编程 (functional programming) 的方式提供交互式的验证环境.

Isabelle/HOL 是一种类型系统 (type system)，类型变量 (type variable) 可以采用 $'a$，$'b$ 等方式来表达；对于类型的项 (term) 如 $x :: 'a$，表示变量 x 是类型 $'a$ 的项. 对于复杂数据类型的构造，可以采用 3 种方式：types，datatype 和 record. types 用于定义数据类型的简化别名，如 types *pid* = int，定义新类型 *pid* 是整数的别名；datatype 用于定义复合的结构类型，如我们对于消息类型定义为

$$\text{datatype } msg = m\_1 \mid m\_2 \mid m\_3 \mid m\_4 \mid m\_5 \mid m\_6 \mid Notify$$

表示 VTOS 的消息类型包括 6 种普通消息格式和通知类型，以 $m\_1 \sim m\_6$ 和 $Notify$ 表示.

record 用于定义带名称的元组类型，如对于 2.2.1 节描述的消息对象，可以定义为

record $message = m\_source :: pid$
$m\_type :: msg$
$m\_content ::$ string

表示消息 $message$ 包含 3 个部分：m_source 表示消息的来源，类型是 pid，即进程标识符；m_type 表示消息的类型，类型是新类型 msg，msg 描述消息的种类，可以使用 datatype 构造；$m\_content$ 表示消息体，类型是字符串 string. 假设 m 的类型为 $message$，引用成员域可以表达为如 $m\_source\ m$，表示引用 m 的 m_source 域. 对于 record 类型的更新操作，假设 m 拥有值($|m\_source = 2, m\_type = m\_1, m\_content =$ "request"$|$)，那么更新操作表达为如 $m(|m\_content := \text{"ack"}|)$，表示对象 m 中 $m\_content$ 域修改为"ack"，其他域保持不变.

对于函数定义，采用"$\Rightarrow$"符号描述函数从定义域到值域的映射关系. 函数更新操作表达为如 $g(x := y)$，表示函数 g 在 x 处的值修改为 y，定义域其他点保持不变.

对于集合操作，可以采用"$\{s.P\ s\}$"定义一个集合，其中所有的元素满足谓词公式 P；对于集合的"补"操作，可以使用"$-A$"来表示集合 A 的补集.

Isabelle/HOL 支持 lambda 演算，例如对于集合运算 $Y = C \cup X$，C 为集合常量，X、Y 为集合变量，可以使用"$\lambda X.C \cup X$"来表示函数 Y 的操作子.

Isabelle/HOL 将关系(relation)作为 2 元对(pair)的集合来看待. 对于关系的转置(converse)定义为：$(a,b) \in M^{-1} \equiv (b,a) \in M$，$M^{-1}$ 表示关系 M 的转置关系；关系的自反传递闭包(reflexive transitive closure)使用如 M^* 来表示. 对于集合在关系上的"象(image)"集合，使用如"$M'' A$"来表示，定义为 $b \in (M'' A) \equiv \exists a \in A.(a,b) \in M$，即"$M'' A$"表示由集合 A 中元素在关系 M 中的 2 元对的第 2 元组成的集合.

下面我们阐述 VTOS 微内核模型 OSOSM 在 Isabelle/HOL 中的建模过程.

4.2　OSOSM 的 Isabelle/HOL 方式建模

4.2.1　对象论域

对于第 2 节描述的 VTOS 微内核对象论域，Isabelle/HOL 方式定义如下：

datatype $rtsflag =$
$RECEIVING \mid SENDING \mid$ NULL
datatype $data = data\_from\_disk \mid data\_from\_input$
types $pid =$ int
types $intr = INTR\_DISK \mid INTR\_KEYBOARD$
record $message = m\_source :: pid$
$m\_type :: msg$
$m\_content ::$ string
record $process = proc\_nr :: pid$
$p\_messbuf :: message$
$p\_rts\_flags :: rtsflag$
$p\_getfrom ::$ "$pid \mid ANY$"
$p\_sendto :: pid$
$p\_lastcall ::$
"$(SEND \mid RECEIVE \mid NOTIFY) \times$
$(ELOCKED \mid OK)$"
$s\_notify\_pending ::$ "pid list"
$s\_msg\_pending ::$ "pid list"
$p\_priority ::$ nat
$p\_ticks\_left ::$ nat
record $state = pset ::$ "$pid \Rightarrow process$"
$process\_q ::$ "nat $\Rightarrow pid$ list"
$sys\_buffer :: data$
$intr\_source :: intr$
$next\_running\_proc :: pid$
$prev\_proc :: pid$
$S ::$ "$state$ set"

其中，$rtsflag$ 为消息发送和接收状态的类型；$intr$ 为中断类型，包括硬盘中断 $INTR\_DISK$ 和键盘中断 $INTR\_KEYBOARD$；$p\_lastcall$ 为进程对象最近一次消息行为语义的标识信息，定义为 2 元对，第 1 元取值 $SEND \mid RECEIVE \mid NOTIFY$，第 2 元取值 $ELOCKED \mid OK$.

4.2.2　VTOS 微内核行为语义

下面我们对第 2 节中描述的 VTOS 微内核行为语义采用 Isabelle/HOL 方式进行表达.

(1) 消息处理行为

在基本功效层，从功效上来说，微内核的消息处理行为是将发送者进程 PCB 对象所指示的消息缓冲区中的消息体复制到接收者进程 PCB 对象所指示的消息缓冲区中. VTOS 消息处理采用"汇合机制"，为此我们使用 Isabelle 描述时需要考虑 3 个状态：消息放送状态、消息接收状态、完成功效状态. 消息、通知发送行为和接收行为具体定义如下：

```
definition SysSend :: "pid => pid => state => state" where
"SysSend p1 p2 s ≡
  (let newp1_1 = (pset s) p1 (| p_lastcall := (SEND, ELOCKED) |);
       newp1_2 = (pset s) p1 (| p_lastcall := (SEND, OK) |);
       newp1_3 = (pset s) p1 (| p_lastcall := (SEND, OK), p_rts_flags := SENDING, p_sendto := p2 |);
       newp2_1 = (pset s) p2 (| p_messbuf := (p_messbuf ((pset s) p1)), p_rts_flags := NULL |);
       newp2_2 = (pset s) p2 (| s_msg_pending := (s_msg_pending ((pset s) p2)) @ (p1 # [] ) |)
   in (    if p_rts_flags ((pset s) p2) = SENDING ∧
          p_sendto ((pset s) p2) = p1 then s (| pset := (pset s)(p1 := newp1_1) |)
      else (if p_rts_flags ((pset s) p2) = RECEIVING ∧
              (p_getfrom ((pset s) p2) = ANY ∨ p_getfrom ((pset s) p2) = p1)
            then SysSchedule p2 (s (| pset := (pset s)(p1 := newp1_2, p2 := newp2_1) |))
            else SysDequeue p1 (s (| pset := (pset s)(p1 := newp1_3, p2 := newp2_2) |)))))"

definition SysNotify :: "pid => pid => state => state" where
"SysNotify p1 p2 s ≡
  (let newp1 = (pset s) p1 (| p_lastcall := (NOTIFY, OK) |);
       newp2_1 = (pset s) p2 (| p_messbuf := (| m_source = p1, m_type = Notify, m_content = NULL |),
                                p_rts_flags := NULL |);
       newp2_2 = (pset s) p2 (| s_notify_pending := (s_notify_pending ((pset s) p2)) @ (p1 # [] ) |)
   in (    if p_rts_flags ((pset s) p2) = RECEIVING ∧
          (p_getfrom ((pset s) p2) = ANY ∨ p_getfrom ((pset s) p2) = p1)
        then SysSchedule p2 (s (| pset := (pset s)(p1 := newp1, p2 := newp2_1) |))
        else s (| pset := (pset s)(p1 := newp1, p2 := newp2_2) |)))"

definition SysReceive :: "pid => pid => state => state" where
"SysReceive p2 p1 s ≡
  (let p1_1 = hd (s_msg_pending ((pset s) p2));
       newp1_1 = (((pset s) p1_1) (| p_rts_flags := NULL |);
       newp1_2 = (((pset s) p1)(| p_rts_flags := NULL |);
       newp2_1 = (pset s) p2 (| p_lastcall := (RECEIVE, OK),
                                p_messbuf := (| m_source = hd (s_notify_pending ((pset s) p2)),
                                                m_type = Notify, m_content = NULL |),
                                s_notify_pending := tl s_notify_pending ((pset s) p2) |);
       newp2_2 = (pset s) p2 (| p_lastcall := (RECEIVE, OK),
                                p_messbuf := p_messbuf ((pset s) (hd (s_msg_pending ((pset s) p2)))),
                                s_msg_pending := tl s_msg_pending ((pset s) p2) |);
       newp2_3 = (pset s) p2 (| p_lastcall := (RECEIVE, OK),
                                p_messbuf := p_messbuf ((pset s) (p_getfrom ((pset s) p2))),
                                s_msg_pending := filter (λx. x ≠ p_getfrom ((pset s) p2))
                                                        s_msg_pending ((pset s) p2) |);
       newp2_4 = (pset s) p2 (| p_lastcall := (RECEIVE, OK), p_rts_flags := RECEIVING |)
   in (if s_notify_pending ((pset s) p2) ≠ []
       then s (| pset := (pset s)(p2 := newp2_1) |)
       else (if s_msg_pending ((pset s) p2) ≠ []
```

```
        then (if p_getfrom ((pset s) p2)=ANY
            then SysSchedule p1_1 (s (|pset := (pset s)(p1_1 := newp1_1, p2 := newp2_2)|))
            else (if p1 mem (s_msg_pending ((pset s) p2))
                then SysSchedule p1 (s (|pset := (pset s)(p1 := newp1_2, p2 := newp2_3)|))
                else SysDequeue p2 (s(|pset := (pset s)(p2 := newp2_4)|))))
        else SysDequeue p2 (s (|pset := (pset s)(p2 := newp2_4)|))))"
```

其中，*SysSchedule* 和 *SysDequeue* 为调度行为，在下面将定义. Isabelle/HOL **中的列表“过滤”操作子** filter **将不满足谓词条件的列表元素删除，语义定义为**

filter P [ ]≡[ ]

filter P (x #xs)≡(if P x then x # filter P xs
else filter P xs)

Isabelle/HOL **中的列表“蕴含”操作子** mem **判断元素是否在列表中，语义定义为**

x mem [ ]≡False

x mem (y #ys)≡(if y=x then True else x mem ys)

(2) **进程调度行为**

微内核的多级进程队列入队操作行为 *schedule* **是将被调度的进程** PCB **对象加入到进程队列对象中.** *dequeue* **行为将进程对象从多优先级进程队列中去除.** *pick_proc* **行为从多优先级进程队列中选择将要运行的进程，根据第** 2 **节的语义描述，**Isabelle/HOL **方式描述如下：**

```
definition SysSchedule :: "pid => state => state" where
" SysSchedule p s ≡
    (let priority = p_priority ((pset s) p);
        newprocess_q_1 = (process_q s)(priority := p # ((process_q s) priority));
        newprocess_q_2 = (process_q s)(priority+1 := (process_q s) (priority+1)@(p # [ ]));
        newprocess_q_3 = (process_q s)(priority-1 := (process_q s) (priority-1)@(p # [ ]));
        newprocess_q_4 = (process_q s)(priority := ((process_q s) priority)@(p # [ ]));
        newp_1 = ((pset s) p) (|p_priority := priority+1|);
        newp_2 = ((pset s) p) (|p_priority := priority-1|);
     in (if p_ticks_left ((pset s) p)≠0 then s (|process_q := newprocess_q_1|)
        else (if p>0 ∧ prev_proc s = p ∧ priority<15
            then s (|process_q := newprocess_q_2, pset := (pset s)(p := newp_1)|)
            else (if p>0 ∧ prev_proc s ≠ p ∧ priority >0
                then s (|process_q := newprocess_q_3, pset := (pset s)(p := newp_2)|)
                else s (|process_q := newprocess_q_4|)))))"

definition SysDequeue :: "pid => state => state" where
" SysDequeue p s ≡ s (|process_q :=
            (process_q s)(p_priority ((pset s) p) :=
                filter (λx.x≠p)((process_q s)(p_priority ((pset s) p))))|)"

definition SysPick_proc :: "state => state" where
" SysPick_proc s ≡ s (|next_running_proc := pick_p s 0 15|)"

primrec pick_p :: "state => nat => nat => nat" where
" pick_p t begin end = (if (process_q t) begin = [ ] ∧ begin < end then pick_p t (begin+1) end
            else (if (process_q t) begin ≠ [ ] then hd ((process_q t) begin)
                else IDLE))"
```

其中，primrec 是 Isabelle/HOL 对原始递归函数的定义.

(3) 中断处理行为

微内核的中断处理行为的功效是根据不同的中断源，对系统缓冲区对象进行设置，本文主要描述键盘中断和磁盘中断. Isabelle/HOL 方式描述如下：

```
definition SysInterrupt_handle :: "state => state" where
"SysInterrupt_handle s ≡
   s (| sys_buffer := (if (intr_source s) = INTR_DISK then data_from_disk
                       else (if (intr_source s) = INTR_KEYBOARD then data_from_input
                             else (sys_buffer s))) |)"
```

4.3 时序逻辑在 Isabelle 中的验证方法

第 3 节采用时序逻辑对 VTOS 微内核的安全需求进行了描述，为了使用 Isabelle/HOL 验证 OSOSM 与安全需求的一致性，首先需要解决 Isabelle/ HOL 验证环境中构建时序逻辑的验证过程.

时序逻辑在模型检测(model checking)中用于描述系统的需求规格说明(specification)，各种模型检测工具都支持时序逻辑的验证. 在定理证明器 Isabelle/HOL 中，使用时序逻辑对被验证的系统进行规格说明并验证的工作，存在如何对时序逻辑进行抽象描述的问题. VTOS 形式化设计首次尝试在 Isabelle/HOL 环境、操作系统级的验证过程中实现对时序逻辑的描述.

下面，我们阐述 VTOS 微内核安全需求的时序逻辑部分在 Isabelle/HOL 中的描述方法.

VTOS 将各种行为的执行功效看成是对系统状态的改变或者迁移，状态转换集合 ST 定义为 ST :: "(state × state) set"，表示系统中状态之间转换的集合，例如 $s2$ 是 $s1$ 的直接后继状态，则二元对 $(s1, s2) \in ST$.

对于第 3 节描述的安全需求的时序逻辑部分中的命题类型，我们定义如下：

```
datatype F = Atomic "atomic" | And F F |
             Or F F | Negative F | AG F | EF F | AX F
```

表示命题公式类型 F 可以是“原子命题 Atomic”、“合取命题 And”、“析取命题 Or”、“否定命题 Negative”以及带时序逻辑的命题 AG/EF/AX.

状态可满足的原子命题集合定义为 *Sat_Atomic* :: "state => atomic set"，状态 s 下可满足的原子命题集合表示为 $Sat\_Atomic\ s$，$Sat\_Atomic\ s \equiv \{f.\ fs = True\}$；

VTOS 微内核行为类型包括消息处理、进程调度和中断处理行为，其定义如下：

```
datatype action = send        pid pid
                | notify      pid pid
                | receive     pid pid
                | schedule    pid
                | dequeue     pid
                | pick_proc   pid
                | interrupt_handle
```

系统行为单步执行引起的状态转换函数 *step* 定义如下：

```
fun step :: "state => action => state" where
  "step s (send p1 p2) = SysSend p1 p2 s" |
  "step s (notify p1 p2) = SysNotify p1 p2 s" |
  "step s (receive p2 p1) = SysReceive p2 p1 s" |
  "step s (schedule p) = SysSchedule p s" |
  "step s interrupt_handle = SysInterrupt_handle s"
```

下面我们定义安全需求命题满足性函数，即状态对安全需求命题的可满足性，和第 3 节保持一致，我们使用“$s \models f$”符号表示“在 s 状态下，命题 f 成立”. 其定义如下：

```
primrec sat :: "state => F => bool"  ("_ ⊨ _") where
  "s ⊨ Atomic a    = (a ∈ Sat_Atomic s)" |
  "s ⊨ And b c     = (s ⊨ b ∧ s ⊨ c)" |
  "s ⊨ Or b c      = (s ⊨ b ∨ s ⊨ c)" |
  "s ⊨ Negative d  = (¬ (s ⊨ d))" |
  "s ⊨ AG f        = (∀w. (s, w) ∈ ST* → w ⊨ f)" |
  "s ⊨ EF f        = (∃w. (s, w) ∈ ST* ∧ w ⊨ f)" |
  "s ⊨ AX f        = (∀w. (s, w) ∈ ST → w ⊨ f)"
```

在第 3 节中，我们对时序逻辑进行了扩展，加入了动作对状态路径分支的影响，在 Isabelle/HOL 中的定义如下：

```
definition step_sat :: "state => action => F => bool"
                       ("_ , _ ⊢ _") where
  "s, a ⊢ f ≡ step s a ⊨ f"
```

接下来我们定义求取满足安全需求命题的状态集的函数：

```
primrec SAT::"F⇒state set" where
"SAT(Atomic a)    ={s.a∈Sat_Atomic s}" |
"SAT(And b c)     =SAT b ∩ SAT c" |
"SAT(Or b c)      =SAT b ∪ SAT c" |
"SAT(Negative d)  =-SAT d" |
"SAT(AG f)        ={s.∀w.(s,w)∈ST*→
                          w∈SAT f}" |
"SAT(EF f)        =lfp(λX.(ST^-1'' X)∪SAT f)" |
"SAT(AX f)        ={s.∀w.(s,w)∈ST→
                          w∈SAT f}"
```

对于上述满足性函数 SAT，原子命题满足性计算即是求对原子命题满足的状态集合；对于“反”命题即是原始命题可满足集合的补集；合取命题即是各命题可满足集合的交集，对应地，析取命题即是各命题可满足集合的合并；对于时序逻辑 AG 命题，即是求状态对象在自反传递闭包 ST^* 的后继状态（直接或者间接后继）的可满足性；对于时序逻辑 AX 命题，即是求直接后继的可满足性.

对于时序逻辑“EF f”命题的满足性问题，首先计算直接满足 f 的状态集，即“$SAT\ f$”，这是基本步（base step）；然后假设已计算得到的结果状态集合为 X，接下来计算这些直接满足 f 的状态的直接前续状态，即“$ST^{-1\prime\prime}\ X$”，这是推演步（deductive step）；最后递归计算这些前续状态，即对 X 抽象后“$\lambda X.(ST^{-1\prime\prime}\ X)\cup SAT\ f$”的最小不动点（Least Fixed Point），Isabelle/HOL 中使用 lfp 函数来计算最小不动点.

对于第 3 节描述的 VTOS 微内核安全需求 $\varphi 1$ 和 $\varphi 2$，是关于时序逻辑“EF f”命题的验证，按照上述“EF f”的 SAT 满足性的计算方法，在验证过程中，我们需要首先计算基本步，这通过构造行为的完成功效状态来实现；然后计算推演步，这通过说明基本步构造的完成功效状态是行为发生状态的后续状态来说明. $\varphi 3$，$\varphi 7$，$\varphi 9$，$\varphi 10$ 是关于时序逻辑“AG f”命题的验证，在验证过程中，首先说明当前状态满足命题 f，然后验证在执行系统行为后的状态仍然能满足命题 f，由此说明从此状态出发的任何状态路径上的所有点都满足命题 f. 如图 4 所示.

$$\frac{f\in Sat\_Atomic\ u \quad (s,u)\in ST^*}{s\models \mathrm{EF}\ f}\ (\mathrm{EF\text{-}intro})$$

$$\frac{f\in Sat\_Atomic\ s \quad \forall u\in S,\ v\in S.(u,v)\in ST\wedge f\in Sat\_Atomic\ u\rightarrow f\in Sat\_Atomic\ v}{s\models \mathrm{AG}\ f}\ (\mathrm{AG\text{-}intro})$$

图 4　VTOS 中时序逻辑 EF/AG 验证方法

4.4　一致性验证

本小节对 VTOS 的设计，即第 2 节阐述 OSOSM 语义模型，是否满足第 3 节的 VTOS 安全需求进行验证，即设计和安全需求的一致性验证.

引理 1.　消息发送行为 *send* 功效完整性. OSOSM 微内核消息发送行为满足 $\varphi 1$ 的功效完整性定义：

```
lemma Send_Effectiveness:
 " ∀s∈S, p1::process, p2::process.
    (s,(send p1 p2))⊢
    EF (λw::state. p_messbuf ((pset w) p2)=
                  p_messbuf ((pset s) p1)"
```

证明.　我们首先将“$s,a\vdash f$”部分使用单步执行函数 *step* 的定义展开，然后构造消息发送行为实现语义函数 *SysSend* 中的接收状态和完成功效状态. 如果在 s 的后继状态 t（直接或者间接后继）执行 *receive p2 p1* 或者 *receive p2 ANY* 行为，这样的 t 即为接收状态，在 t 的直接后继状态 w 时，按照 *SysSend* 函数的定义，完成 *SysSend* 的功效，因此 w 即为完成功效状态. Isabelle/HOL 证明过程如下：

```
apply(simp add: step_sat_def sat_def SysSend_def);
apply(subgoal_tac" w=step t (receive p2 p1)");
apply(blast);
apply(simp add: SysReceive_def);
apply(auto);
done;
```

其中，apply 表示使用规则或者已证明的引理和定理来进行验证；simp 方法对“$s,a\vdash f$”部分使用单步满足性的定义（*step_sat_def*）和系统行为定义（*SysSend_def*）来进行展开并化简；subgoal_tac 方法使用“$w=step\ t\ (receive\ p2\ p1)$”构造接收状态和完成功效状态；blast 表示使用经典推理方法（classical reasoning methods）进行前向（forward）搜索验证；auto 表示对验证目标进行简化，并根据已有结论和定义进行搜索验证.　　证毕.

采用类似的基本步和推演步的构造方法，我们可以证明通知发送行为和接收行为的功效完整性.

引理 2. **通知发送行为 *notify* 功效完整性.** OSOSM 微内核通知发送行为满足 $\varphi 2$ 的功效完整性定义：

```
lemma Notify_Effectiveness:
" ∀s∈S, p1::process, p2::process.
   (s,(notify p1 p2)) ⊢
     EF (λw::state. p_messbuf ((pset w) p2) =
                    (p1, Notify, NULL))"
```

定理 1. **消息处理行为完整性**(Message Process Integrity, MPI). **假设系统初始状态 *s*0 为可信状态，在 *s*0 的将来任何状态下，微内核都具有消息发送行为功效完整性、通知发送行为功效完整性和接收行为功效完整性：**

```
theorem MPI:
 "s0 ⊨ AG ((∀s∈S, p1::process, p2::process.
   (s,(send p1 p2)) ⊢
    EF (λw::state. p_messbuf ((pset w) p2) =
                   p_messbuf ((pset s) p1)) ∧
   (∀s∈S, p1::process, p2::process.
   (s,(notify p1 p2)) ⊢
   EF (λw::state. p_messbuf ((pset w) p2) =
                  (p1, Notify, NULL))))"
```

证明. **从引理 1，引理 2 可知，**VTOS **在完成微内核消息处理行为功效的同时，不会破坏完成功效状态下继续进行消息处理的能力，因此在系统初始状态 *s*0 为可信状态的情况下，任何后续状态都能保证消息处理行为的功效完整性.**

```
apply(simp add: step_sat_def sat_def)
apply(blast intro:
        Send_Effectiveness Notify_Effectiveness)
apply(auto)
done
```

证毕.

引理 3. **多级进程队列入队操作行为 *schedule* 功效完整性.** VTOS **微内核的多级进程队列入队操作行为 *schedule* 能正确地将进程加入到多级进程队列中：**

```
lemma Schedule_Effectiveness:
" ∀s∈S, p::process.
   (s,(schedule p)) ⊢
     AX (λw::state. ∃i.(0≤i ∧ i≤15 ∧
                  p mem ((process_q w) i)))"
```

证明. **从 *SysSchedule* 的定义可知，**VTOS **微内核根据被调度进程对象的剩余时间片和** CPU **资源的占用情况来设置进程对象的优先级，并将进程对象加入到新优先级对应的进程队列中，为此在完成功效状态下，进程对象必定在优先级从** 0 **到** 15 **的多级进程队列中.** Isabelle/HOL **证明过程如下：**

```
apply(simp add:
  step_sat_def sat_def SysSchedule_def)
apply(blast)
apply(erule mem_def)
apply(best)
done
```

其中“erule *mem_def*”**表示利用** Isabelle/HOL **中** list **的** mem **函数定义来进行消除规则**(elimination rules)**的前向推导；**best **方法表示采用最优查找**(best-first search)**代替经典推理方法的深度查找**(depth-first search)**进行搜索验证.** **证毕.**

采用类似的方法，我们可以证明多级进程队列出队操作行为 *dequeue* 的功效完整性和选择进程对象行为 *pick_proc* 的功效完整性.

引理 4. **多级进程队列出队操作行为 *dequeue* 行为功效完整性. 微内核的进程队列出队操作行为 *dequeue* 能正确地将进程从多级进程队列中删除：**

```
lemma Dequeue_Effectiveness:
" ∀s∈S, p::process.(s,(dequeue p)) ⊢
     AX (λw::state. ∀i.(0≤i ∧ i≤15 →
                ¬(p mem ((process_q w) i))))"
```

引理 5. ***pick_proc* 行为功效完整性.**

```
lemma Pick_proc_Effectiveness:
" ∀s∈S. (s, pick_proc ⊢
     AX (λw::state.(next_running_proc w) =
                  pick_p s 0 15))"
```

定理 2. **进程调度行为完整性**(Process Dispatch Integrity, PDI). **假设系统初始状态 *s*0 为可信状态，在 *s*0 的将来任何状态下，微内核都具有多级进程队列入队操作行为功效完整性、多级进程队列出队操作行为功效完整性和选择进程对象行为功效完整性.**

```
theorem PDI:
 "s0 ⊨ AG
   (∀s∈S, p::process.(s,(schedule p)) ⊢
     AX (λw::state. ∃i.(0≤i ∧ i≤15 ∧
       p mem ((process_q w) i)))) ∧
   (∀s∈S, p::process.(s,(dequeue p)) ⊢
     AX (λw::state. ∀i.(0≤i ∧ i≤15→
      ¬(p mem ((process_q w) i)))))) ∧
```

(∀s∈S.(s, $pick\_proc$ ⊢
AX (λw :: $state$.($next\_running\_proc$ w) =
$pick\_p$ s 0 15)))"

证明. 从引理 3、引理 4、引理 5 可知，VTOS 在完成进程调度行为功效的同时，不会破坏完成功效状态下继续进行进程调度的能力，因此在系统初始状态 $s0$ 为可信状态的情况下，任何后续状态都能保证进程调度行为的完整性. 证毕.

类似地，我们有中断处理行为的功效完整性.

定理 3. 中断处理行为完整性(Interrupt Handle Integrity，IHI). 假设系统初始状态 $s0$ 为可信状态，在 $s0$ 的将来任何状态下，VTOS 的中断处理行为能正确地根据不同的中断源，对系统缓冲区对象进行设置：

```
theorem IHI:
"s0 ⊨ AG (∀s∈S.(s, interrupt_handle ⊢
    AX (λw :: state.
      ((intr_source s) = INTR_DISK →
        (sys_buffer w) = data_from_disk) ∨
      ((intr_source s) = INTR_KEYBOARD →
        (sys_buffer w) = data_from_input)))"
```

证明. 使用单步执行函数 $step$ 展开后，根据 $SysInterrupt\_handle$ 的定义，VTOS 微内核对中断的处理将根据中断源的类型如硬盘中断、键盘中断，进行设置系统缓冲区.

```
apply(simp add:
    step_sat_def sat_def SysInterrupt_handle_def)
apply(case_tac intr_source)
apply(auto)
done
```

其中“case_tac $intr\_source$”表示对中断源变量 $intr\_source$ 进行分情况展开. 证毕.

我们的验证环境配置如表 1 所示. VTOS 微内核部分的 Isabelle/HOL 验证工程代码量大概在 23k SLOC(Source Lines Of Code)左右，完整的验证耗时 15 min 左右.

表 1　验证系统配置信息

名称	版本	配置
Hardware	Dell Studio XPS 9100	Standard Installation
CPU	Intel i7 930	2.8 GHz
Memory	DDR3 SDRAM	3 G
OS	openSUSE Desktop 11.3	Standard Installation
Isabelle	Isabelle2009-2_bundle_x86-linux	Standard Installation

Isabelle 的验证结果如图 5 所示. “No subgoals”说明 Isabelle 验证逻辑完整，不存在任何未证明的子目标.

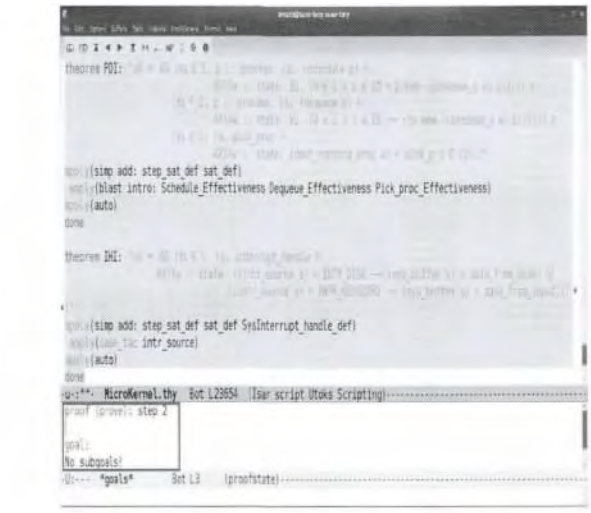

图 5　Isabelle 验证结果

5　结束语

本文使用形式化方法对操作系统进行设计和验证，阐述在 VTOS 设计过程中构建对象语义模型 OSOSM 的方法，以此来对 VTOS 的设计进行形式化的描述. 本文从系统行为功效完整性和进程行为隔离性的角度对微内核 OS 的安全需求进行了分析，使用带有时序逻辑的高阶逻辑对安全需求进行严格的定义和描述，并在定理证明器 Isabelle/HOL 环境中对系统的设计和安全需求的一致性进行验证，表明 VTOS 微内核系统行为的功效设计符合安全需求的定义.

本文采用的对象语义模型 OSOSM 是一种开放的框架模型，可以实现对 OS 内核以及功能模块的描述. 由于系统的各种功能模块往往采用多种不同的程序逻辑进行设计，并且涉及多种不同的抽象层次，如 C 语言层、汇编语言层和硬件实现层等，对于这些功能模块整合在一起的系统的正确性不能简单地认为是各个模块的正确性的合取. 我们接下来的工作将从域理论(domain theory)和类型论(type theory)的角度来对系统的各个经过验证的模块的整合验证进行研究.

致　谢　本文作者感谢本文的所有匿名审稿者！

参　考　文　献

[1] Klein G, Andronick J, Elphinstone K, et al. seL4: Formal verification of an operating-system kernel. Communications of the ACM, 2010, 53(6): 107-115

[2] Alkassar E, Hillebrand M A, Leinenbach D, et al. The Verisoft approach to systems verification//Proceedings of the 2nd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2008). Toronto, Canada, 2008: 209-224

[3] Stampoulis A, Shao Z. Static and user-extensible proof checking//Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012). Philadelphia, USA, 2012: 273-284

[4] Elphinstone K, Heiser G. From L3 to seL4 — what have we learnt in 20 years of L4 microkernels?//Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2013). Farmington, USA, 2013: 133-150

[5] O'Sullivan B, Stewart D, Goerzen J. Real World Haskell. California: O'Reilly, 2008

[6] Nipkow T, Paulson L C, Wenzel M T. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Heidelberg: Springer, 2002

[7] Sewell T, Winwood S, Gammie P, et al. seL4 enforces integrity//Proceedings of the 2nd Conference on Interactive Theorem Proving (ITP 2011). Nijmegen, Netherlands, 2011: 325-340

[8] Heiser G, Murray T, Klein G. It's time for trustworthy systems//Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P 2012). San Francisco, USA, 2012: 67-70

[9] Alkassar E, Cohen E, Hillebrand M A, et al. Verifying shadow page table algorithms//Proceedings of the 10th International Conference on Formal Methods in Computer Aided Design (FMCAD 2010). Lugano, Switzerland, 2010: 267-270

[10] Baumann C, Bormer T, Blasum H, et al. Proving memory separation in a microkernel by code level verification//Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW 2011). Newport Beach, USA, 2011: 25-32

[11] Vaynberg A, Shao Z. Compositional verification of a baby virtual memory manager//Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP 2012). Kyoto, Japan, 2012: 143-159

[12] Liang Hong-Jin, Feng Xin-Yu, Fu Ming. A rely-guarantee-based simulation for verifying concurrent program transformations//Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012). Philadelphia, USA, 2012: 455-468

[13] Qian Zhen-Jiang, Liu Wei, Huang Hao. OSOSM: Operating system object semantics model and formal verification. Journal of Computer Research and Development, 2012, 49(12): 2702-2712(in Chinese)
(钱振江, 刘苇, 黄皓. 操作系统对象语义模型(OSOSM)及形式化验证. 计算机研究与发展, 2012, 49(12): 2702-2712)

[14] Långbacka T. A HOL formalization of the temporal logic of actions//Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications. Valletta, Malta, 1994: 332-347

[15] Benthem J V, Doets K. Higher-order logic//Gabbay D M, Guenthner F, eds. Handbook of Philosophical Logic: Volume 1. Dordrecht: Kluwer Academic Publishers, 2001: 189-243

[16] Zhou Chao-Chen. Introduction to Formal Semantics. Beijing: Institute of Computing Technology, Chinese Academy of Sciences, 1985(in Chinese)
(周巢尘. 形式语义学引论. 北京: 中国科学院计算技术研究所, 1985)

QIAN Zhen-Jiang, born in 1982, Ph. D., lecturer. His current research interests include operating system security, formal verification and embedded systems.

HUANG Hao, born in 1957, Ph. D., professor, Ph. D. supervisor. His current research interests include system software and information security.

SONG Fang-Min, born in 1961, Ph. D., professor, Ph. D. supervisor. His current research interests include symbolic logic and quantum computers.

Background

This work is mainly supported by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China under Grant No. 60721002, 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, and CSLG Science Research Program No. QT1312. They all aim to develop a secure and trusted microkernel operating system. The group has studied novel security technologies and mathematical formal methods. The group also studied formal specifications and descriptions of the designed microkernel operating system, and strictly

verified the correctness of its implementation. In past years, the group has done a lot of related works including a trusted boot revolution, a verified lightweight approach to provide lifetime kernel integrity surveillance (HybridHP), an operating system object semantics model (OSOSM) and its partial formal specifications and verification with Isabelle/HOL theorem prover.

VTOS microkernel operating system related in this paper is one of the important achievements of State Key Laboratory for Novel Software Technology of Nanjing University in the period of 2008—2013, and is also the software product which is accepted in the assessment for State Key Laboratory. The object semantics model OSOSM emphasizes that the formal methods should be used in the early design process for VTOS, in order to guarantee the verifiability of the whole system in the design process, and strive for "full" guidance of formal methods and theories to the design of the whole system.

In the paper "Qian Zhen-Jiang, Liu Wei, Huang Hao. OSOSM: Operating system object semantics model and formal verification", the authors proposed and illustrated the OSOSM model, and took the VMM module in VTOS as an example to explain the specification and verification of isolation property of process objects. From the aspect of microkernel of VTOS, this paper describes the system design and analysis of security requirements, and mainly elaborates the problems in the design process of VTOS with OSOSM. Meanwhile, this paper illustrates the formal description and strict definition of security requirements, such as integrity of system effectiveness and isolation of process behavior. This paper also uses Isabelle/HOL theorem prover to verify the consistency between system design and security requirements, and shows that VTOS achieves the desired security.