基于不干扰理论的隔离语义描述及隔离策略的自动化验证方法研究

崔 隽[1,2] 黄 皓[1,2] 陈志贤[1,2,3]

(南京大学软件新技术国家重点实验室 南京 210093)[1] (南京大学计算机科学与技术系 南京 210093)[2]

(南京工业大学信息学院 南京 210009)[3]

摘 要 隔离有助于阻止信息泄露或被篡改、错误或失败被传递等。利用不干扰理论给出了隔离的精确语义，以利于分析和制定系统的隔离策略；利用通信顺序进程CSP来定义上述隔离语义，并给出一个系统满足给定隔离策略的判定断言，以利于借助形式化验证工具FDR2来实现系统内隔离策略的自动化验证。以基于虚拟机的文件服务监控器为例，展示了如何利用CSP来建模一个系统及其隔离策略以及如何利用FDR2来验证该系统模型满足给定的隔离策略。

关键词 不干扰模型，进程隔离，通信顺序进程，形式化验证

中图法分类号 TP301 **文献标识码** A

Research on Isolation Semantics Description Based on Noninterference Theory and Automated Isolation Strategy Verification Scheme

CUI Jun[1,2] HUANG Hao[1,2] CHEN Zhi-xian[1,2,3]

(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)[1]

(Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)[2]

(College of Information Science and Engineering, Nanjing University of Technology, Nanjing 210009, China)[3]

Abstract Processes or modules isolation helps protect information from being revealed or modified and prevent processes or modules from passing error or failure to others. We proposed the semantics of isolation by noninterference theory, for the purpose of analyzing and designing isolation strategies in software systems; we also specified the semantics of isolation and its determine conditions by Communicating Sequential Process(CSP) in order for automated formal verification of isolation strategies in systems in formal verification tool FDR2. And in this paper, with an example of file system monitor in a virtual machine, we illustrated how to specify a system or a isolation strategy by CSP formulation and how to verify given isolation strategies in a system automatically in FDR2.

Keywords Noninterference model, Processes isolation, Communicating sequential processes, Formal verification

1 引言

进程或模块的隔离有利于阻止系统的关键数据或代码受到其它恶意进程的干扰或破坏，有利于阻断错误或失败在进程或模块之间蔓延。隔离属性几乎是所有软件系统或网络服务中都不可或缺的基本安全属性。隔离又分两种情况：一种如图1(a)所示，隔离进程或模块之间不存在任何共享资源；另一种如图1(b)所示，虽然隔离进程或模块之间存在共享的资源，但在同一时刻，任意资源只能被一个进程或模块访问，并且任意资源在被一个主体(进程或模块)使用结束，被另一个主体使用之前，该资源所承载的信息会被清除。其中一种更复杂的情况是，隔离主体共享的可能是服务(如图2(a)所示)，而不是简单的资源，但与共享资源类似，共享服务拥有的资源同一时刻只能被一个服务响应线程使用，且每次使用后信息都会被清除。这样的隔离虽不直观，却普遍存在于基于C/S或B/S架构的软件系统中，用于保护客户数据的完整性，维护客户信息的隐私性。因此，需要合适的信息流策略来描述这种隔离关系。现有的信息流策略，如BLP、RBAC、不干扰策略等都仅关注两个信息域主体之间的直接关系，而并不关心这两个信息域是否通过共享的第三方进程或模块传递信息。即使是非传递的不干扰策略模型，它也比其它的信息流策略更注重信道控制。如图2(b)所示，非传递的不干扰策略允许信息域A直接干扰B，也允许B直接干扰C，但由于策略是非传递的，因此除非策略明确定义，否则A直接干扰C是不被允许的，这就保证了A必须通过B才能间接地影响C，B就如同A与C之间的信道。对比之前分析的隔离的第二种情形，非传递的不干扰策略仍无法描述如图2(a)的策略需求，因为不干扰策略中默认B是A与C之间的信道，即不能

到稿日期：2009-09-20 返修日期：2009-10-27 本文受863国家高技术研究发展计划(No：2007AA01Z409)资助。

崔 隽(1981－)，男，博士生，主要研究方向为操作系统形式化验证、安全模型，E-mail：ctops@sina.com；**黄 皓**(1957－)，男，教授，博士生导师，主要研究方向为网络安全；**陈志贤**(1979－)，男，博士后，讲师，主要研究方向为网络安全、数据挖掘。

明确定义 B 隔离 A 和 C 之间信息交互的需求。而图 2(a)中，B 不被允许在 A 和 C 之间传递信息。所以，需要考虑新的信息流策略，使得能够同时描述图 1(a)和图 2(a)的两种隔离策略。而图 1(b)中共享资源的情形，如果把共享资源及资源操作看做数据服务的话，则类似于图 2(a)的信息流策略。本文在深入分析了上述两种隔离情形的基础之上，给出了一种描述执行主体隔离特性的新的信息流策略，称作隔离策略，记作 $\perp$。如果隔离策略不允许两个信息域通信，则它们既可能如图 1(a)所示不存在任何共享，也可能如图 1(b)或图 2(a)所示，虽然存在共享的资源或服务，但共享资源和服务不能为它们之间传递任何信息。

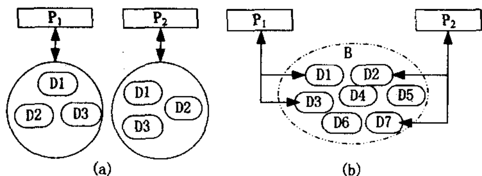

图 1 隔离进程

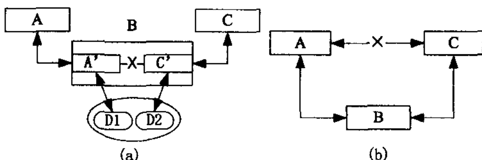

图 2 共享服务的进程

本文隔离策略的研究基础是非传递的不干扰理论。在 1982 年 *Goguen* 和 *Meseguer* 提出的不干扰理论[1]基础上，*Haigh* 和 *Young* 提出了非传递不干扰理论[2]，其主要用于分析和控制信息流通道、处理安全级别降低等其它信息流策略不能很好解决的问题。本文正是基于非传递不干扰理论，研究干扰关系不通过共享第三方传递的隔离策略。而针对不干扰策略的形式化验证，*Roscow* 和合作者[3,4]基于确定性重新给出了不干扰属性的定义，并根据此研究分别给出了验证传递不干扰和非传递不干扰理论的方案[5]。本文仍沿用该思想，并根据隔离策略的特点，设计了适合隔离策略验证的解决方案。

2 非传递不干扰理论

Rushby 在文献[6]中将系统定义为一个有限状态自动机，并基于此定义给出了非传递不干扰关系的基本定义。

定义 1 系统 M 包括以下元素：

- 系统状态集合 S，系统的初始状态记为 s_0。
- 系统动作集合 A。
- 系统输出集合 O。
- 单步状态转换函数 $step: S\times A\rightarrow S$。描述执行动作 A 前后的状态转换。
- 系统输出函数 $output: S\times A\rightarrow O$。描述执行动作 A 后的系统输出，与系统状态相关。
- 系统信息域集合 D
- 系统执行函数 $run: S\times A^{*}\rightarrow S$。描述执行一系列动作后的状态转变。
- 行为执行域函数 $dom: A\rightarrow D$，$dom(a)$ 表示执行行为 a 的信息域。

基于自动机，Rushby 定义了 $D\times D$ 上的不干扰关系 $\rightsquigarrow$ 及其补关系 $\not\rightsquigarrow = D\times D\backslash \rightsquigarrow$。系统的不干扰策略可以表达为系统中不同信息域之间的不干扰关系。为了能够验证一个系统设计是否满足给定的不干扰策略，*Rushby* 给出了满足给定不干扰策略的系统 M 的如下定义。

定义 2 给定系统 M，以及安全策略 $\rightsquigarrow$，如果 $\forall a\in A$，动作序列 $\beta\in A^{*}$ 满足 $output(run(s_0,\beta),a)=output(run(s_0,ipurge(\beta,dom(a))),a)$，则称系统 M 满足安全策略 $\rightsquigarrow$。其中，$ipurge$ 的定义见文献[6]。

由于定义 2 中对系统 M 满足安全策略的判定依赖于对提取函数 $ipurge$ 的计算，而 $ipurge$ 的计算又需要作用于所有的系统执行序列，即使序列是有穷的，人工判定该条件也是非常低效和不现实的。因此，Rushby 又提出了等价于该判定条件的单步展开条件[6]。单步条件虽然避免了对 ipurge 的计算，易于判定，但仍需人工地对每一个系统的执行动作逐一判断，无法适用于大型的复杂系统验证过程。

为此，Roscoe[5]在定义 2 的基础上，基于 CSP[7]规范重新给出了系统 M 满足策略 $\rightsquigarrow$ 的条件。为便于描述，首先给出一些常用的 CSP 描述如下：

$\alpha(P)$ P 执行的动作集合

$a\rightarrow P$ 前缀，if a then P

P/s 后继，P after s

$P\,||\,Q$ 并发，P in parallel with Q

$P\backslash C$ 屏蔽，P with C hiding

$P\uparrow C$ 约束，s restricted to A

$P\,|||\,Q$ 穿插，P interleave Q

$c!\ v, c?\ v$ 输入输出，channel c with message v

$SKIP, STOP$ 空操作和终止操作

$CHAOS(P)$ 不确定选择执行 P 的任意动作的进程

$Traces(P)$ P 可能执行的执行序列的集合

$Failures(P)$ 由 (s,X) 构成，描述 $\forall s\in Traces(P), x\in X$，$s$ 拒绝执行 x

$P=_{f}Q$ 如果 $Failures(P)\subseteq Failures(Q)\wedge Failures(P)\subseteq Failures(Q)$

于是，Roscoe 基于 CSP 规范重新阐述了定义 2。

定义 3 给定确定性系统 M 以及安全策略 $\rightsquigarrow$，如果 $\forall a\in A, u\in D, s, s'\in traces(M)$，满足 $s\backslash\alpha(noflow(u))=s'\backslash\alpha(noflow(u))\Rightarrow head(M/s)\cap\alpha(u)=head(M/s')\cap\alpha(u)$，则称系统 M 满足安全策略 $\rightsquigarrow$。其中，$noflow(u)=\{v\,|\,v\in D, v\not\rightsquigarrow u\}$，$head(M/s)$ 为 M 在执行 s 后可能执行的动作集合。

$head(M/s)\cap\alpha(u)$ 表达了在系统执行轨迹 s 之后立即可以执行的进程 u 上的动作集合。在 CSP 中，即使相同的输入或输出动作，如果输入或输出值不同，也会被视作不同的动作。所以定义 3 中 $head(M/s)\cap\alpha(u)$，不仅能够包含定义 2 中 $output()$ 所表达的进程 u 的输出变化，还可以表达进程 u 在动作执行上发生的改变。定义 3 指出所有 $noflow(u)$ 中进程的执行都不会影响 u 上的输出和下一动作的执行。Roscoe 基于 CSP 给出的该条件可以借助成熟的形式化验证工具来验证，比定义 2 更容易判定系统 M 是否满足给定的不干扰策略。

3 基于不干扰理论的隔离策略研究

3.1 隔离的定义和策略描述

进程或模块的隔离也可以看作是一种不干扰，但是与传

统不干扰的语义不同，隔离表达的不干扰是绝对的，是不会经由任何共享的信道而发生联系的，即一但策略要求两个进程或模块隔离，它们就是互相不可见的。传统的不干扰理论对于任意的信息域 $u,v,w\in D$，如果有 $u\rightsquigarrow v$，$v\rightsquigarrow w$，传统不干扰理论（包括非传递不干扰理论）认为域 u 的所有信息都允许经由 v 流向 w，即 v 的作用被弱化为一个纯粹的信息通道。但是，实际中很多情况并非如此，比如图 3 中的服务进程，它与每一个请求服务的应用进程通信，但是不被允许将一个应用进程的信息透露给另一个应用进程。此时，v 并不是充当联系 u 和 w 的信息通道，u 和 w 传送给 v 的数据在 v 中是隔离的。所以，在本文中，如果 u 和 w 是隔离的，则 u 不仅不能直接干扰 w，也不能透过其他任意进程（如 v）传递信息。这类策略是传统不干扰策略所不能描述的。

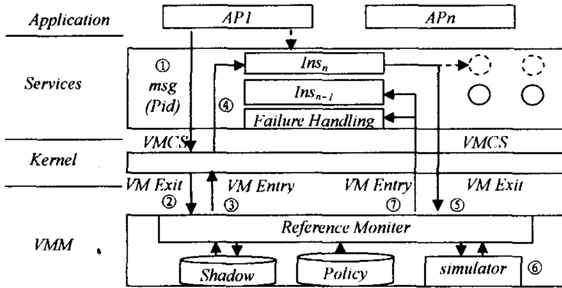

图 3　虚拟机监控流程

为了能够描述隔离策略，本文在传统不干扰理论的基础上，定义了 $D\times D$ 上的新的干扰关系 $\succ$ 及其补关系 $\not\succ=D\times D\backslash\succ$，且 $\succ$ 不具有传递性。对于任意信息域 u,v 上的新干扰关系 $u\succ v$，允许 u 的信息流向 v，并不区分信息流是直接从 u 流向 v，还是经过其它公共第三方信息域传递。同样，非干扰关系 $\not\succ$ 也不区分信息流动的方式，而将阻止所有的直接或间接的信息干扰。新干扰关系 $\succ$ 本身反映的是隔离的特性，不区分信息传递途径，因此，不能解决信道控制等问题，但是可以复合传统的不干扰策略来共同解决。这不是本文讨论的重点，不再累述。

$\succ$ 关系不仅考虑直接的干扰，也考虑由中间进程间接产生的干扰，因此，与传统不干扰模型不同，任意一个行为对其它信息域可能产生的干扰不一定会立刻反映在被干扰的信息域上，可能需要通过多个信息域传递。所以要分析某个域上 u 的行为是否会对另一个域 v 产生影响，仅仅观察 u 的动作在执行结束时刻立即在 v 上产生的输出 $output$ 是不够的，还需观察该行为执行之后 v 在整个生命周期内可能出现的任何与该动作相关的 $output$ 输出。于是，需要扩展定义 1 中仅能反映某个即时状态 s 下目标域上的输出视图 $output$，使其能够反映状态 s 之后，目标域在生命周期内的任意状态下所有可能的输出。另一方面，在实际系统中，动作的执行存在着继承关系。这就要求不干扰理论中任意动作的执行不仅仅与执行进程相关，也必须与执行前的状态相关。因此，还需为系统 M 增加判断任意状态下任意动作是否可执行的描述。于是扩展定义 1 如下。

定义 4　系统 M 除包含定义 1 所有元素外还包括：

- 动作可执行判定函数：$enabled: S\times A\rightarrow Boolean$，$enabled(s,a)$ 表示在状态 S 下动作 a 是否可执行。
- 序列可执行判定函数：$enableds: S\times A^*\rightarrow Boolean$，$enableds(s,\gamma)$ 表示在状态 S 下序列 γ 是否可执行。
- 域输出 $opurge: O\times D\rightarrow O$，$opurge(o,u)$ 描述输出 o 在域 u 上的映射。
- 域执行 $tpurge: A^*\times D\rightarrow A^*$，$tpurge(\alpha,u)$ 描述轨迹 α 在域 u 上的执行序列。
- 域视图：$views: S\times D\rightarrow 2^{(A^*\times O)}$。$views(s,u)=\{(tpurge(\alpha,u),opurge(output(run(s,\alpha)),u))\mid\alpha\in A^*,enableds(s,\alpha)\}$，描述域 u 上执行任意动作序列后的状态视图。

新的域视图 $views(s,u)$ 是一个集合，它由一系列的二元组构成。在给定状态 s 下，每一个二元组都描述了在状态 s 下域 u 执行的任意一个行为序列之后的 $output$ 输出。因此，整个集合即构成了状态 s 之后，域 u 上所有可能的执行序列及其输出。如果 $views(s,u)$ 的两个二元组元素拥有相同的第一分量即执行序列，但是输出不同，则说明存在 u 以外的信息域行为对 u 上输出值的影响。同时，每个集合元素的第一个分量的集合也反映了状态 s 后 u 可能执行的所有动作序列。于是，基于新的视图定义，可以给出满足新不干扰策略 $\not\succ$ 的系统 M 的定义。

定义 5　给定系统 M，以及不干扰策略 $\not\succ$，如果对于任意 $u\in D,\alpha,\beta\in A^*$，满足 $enableds(s_0,\alpha)\wedge enableds(s_0,\beta)\wedge apurge(\alpha,u)=apurge(\beta,u)\Rightarrow views(run(s_0,\alpha),u)=views(run(s_0,\beta),u)$，则称系统 M 满足不干扰策略 $\not\succ$。

其中，提取函数 $apurge\ A^*\times D\rightarrow A^*$ 对于任意 $u\in D$ 有定义：

$$apurge(\Lambda,u)=\Lambda$$

$$apurge(a\circ\beta,u)=\begin{cases}a\circ apurge(\beta,u) & \text{if } dom(a)\succ u\\ apurge(\beta,u) & \text{otherwise}\end{cases}$$

$apurge$ 提取的是所有能够直接或者间接干扰域 u 的动作。定义 5 表达了如果策略 $\not\succ$ 不允许信息从一个域直接或间接流向另一个域，则前者任意行为（包括行为序列）的执行都不会对后者的执行或者输出产生影响。

与定义 2 类似，$apurge$ 以及 $views$ 视图都不便于人工的计算和验证，因此，正如 Roscow 借助于 CSP 为 $ipurge$ 给出进程代数的语义，本文也给出 $apurge$ 以及 $views$ 视图的进程代数语义。于是，定义 4 可以有如下描述。

定义 6　给定系统 M，以及不干扰策略 $\not\succ$，如果对于任意 $u\in D,s,s'\in traces(M)$，满足，$s\backslash\alpha(noflow(u))=s'\backslash\alpha(noflow(u))\Rightarrow(M/s)\upharpoonright\alpha(u)=_f(M/s')\upharpoonright\alpha(u)$，则称系统 M 满足不干扰策略 $\not\succ$。

如同定义 3 中，状态 s 下可执行动作集合 $head(M/s)\cap\alpha(u)$ 能够包含 $output$ 的语义信息一样，状态 s 下可执行动作序列集合 $(M/s)\upharpoonright\alpha(u)$ 也能够包含 $views(s.u)$ 的所有语义信息，同时将可执行序列与 $output$ 输出一同作为状态表征的视图元素，比 $output$ 更能反映干扰对信息域的影响。而 CSP 的失败等价 $=_f$ 也恰恰可以用来比较在 M 执行了行为序列 s 或 s' 之后是否还能执行相同的执行序列。基于定义 6 的不干扰策略，就可以定义信息域的隔离关系语义如下。

定义 7　给定系统 M，以及不干扰策略 $\not\succ$，如果存在信息域 $u,v\in D$，同时满足 $u\not\succ v$ 以及 $v\not\succ u$，则称信息域 u,v 是隔离的，记作 $u\perp v$。

显然，隔离关系是对称的、传递的。两个隔离的信息域之间是不存在任何直接或间接干扰的。由于隔离关系又可以理解为两个隔离信息域之间相互的不干扰关系 $\not\succ$，因此讨论和

验证隔离策略只需针对不干扰关系$\not\rightsquigarrow$进行即可。

3.2 不干扰$\not\rightsquigarrow$验证条件的约简

根据定义 7，就可以判定系统中任意两个信息域是否是隔离的，当然，这个工作不能是人工的，我们用 CSP 重新定义不干扰条件的目的就是要利用一些形式化验证工具（如 FDR2[8]）去自动的验证。然而，定义 6 中，s 是系统 M 从初始状态开始可执行的任意轨迹，因此需要验证的 s 的数量可能会相当大。为了提高验证的效率，将对 s 做一些限制。

首先，在一个实际系统中，可以把任意信息域都描述成进程，而将进程间的同步消息机制描述为进程间的唯一通信方式。系统中原有的异步通信或共享内存的通信方式可以通过构建一个或多个第三方进程来暂存异步消息或操作共享内存数据。而这些第三方进程同样按照同步通信的方式与原系统进程交互。基于 CSP 的描述方法正是利用这种思想来描述一个实际的复杂系统，可以参见文献[7]。而且本文也正是使用 CSP 来描述系统和验证隔离性的。于是，可以将任意进程 u 上的动作划分为两类：同步通信动作集合 $sig(u)$ 和内部动作集合 $inner(u)$。通信动作一定是成对出现的，并且是由通信的发起方和接收方连续执行自己的输入输出动作来实现信息传递或同步，可认为通信接收的动作总是紧接在通信发起动作之后执行的。而内部动作因为无需和其它进程同步，它的执行对其它进程是透明的，也不可能对其它进程有任何直接的影响，即使有任何影响，也要通过通信动作间接地将影响传递出去。因此，可以推出以下性质。

性质 1 $\forall s,s'\in traces(M), s\backslash\alpha(u)=s'\backslash\alpha(u), tail(s)=tail(s')\Rightarrow s\backslash inner(u)=s'\backslash inner(u)$，其中，$tail(s)$ 表示 s 的最后一个动作。

证明：$s\backslash\alpha(u)=s'\backslash\alpha(u)$ 表明 s,s' 在进程 u 以外执行了相同的动作，如果 u 在 s,s' 执行过程中有与其它进程通信，且与之对应的由其它进程执行的通信动作也在 s,s' 中，则在 s,s' 中分别出现的一定是相同的通信动作，否则一定在 s,s' 以外。如果是后者，由于通信进程执行同步通信动作的过程是连续的，则 $tail(s)$ 或 $tail(s')$ 一定是 u 上的通信动作，而 $tail(s)=tail(s')$，则 s,s' 在进程 u 内执行的通信动作一定是一致的，即 $s\backslash inner(u)=s'\backslash inner(u)$ 成立。

性质 2 $\forall s,s'\in traces(M), s\backslash noflow(u)=s'\backslash noflow(u), tail(s)=tail(s'), (M/s)\uparrow\alpha(u)\neq_f(M/s')\uparrow\alpha(u)\Rightarrow\exists t\in traces(M/s), t'\in traces(M/s'), tail(t), tail(t')\in sig(noflow(u))\cup SKIP, t\backslash noflow(u)=t'\backslash noflow(u), tail(t)\neq tail(t'), (M/(s\hat{\ }t))\uparrow\alpha(u)\neq_f(M/(s'\hat{\ }t'))\uparrow\alpha(u)$。

证明：根据 $s\backslash noflow(u)=s'\backslash noflow(u), tail(s)=tail(s')$，由性质 1 知 s,s' 的执行对 $noflow(u)$ 以外的进程是没有直接影响的，$noflow(u)$ 未执行不同的对外通信动作。如果仍有 $(M/s)\uparrow\alpha(u)\neq_f(M/s')\uparrow\alpha(u)$，那是 s,s' 在 $noflow(u)$ 上执行了不同的内部动作导致的间接影响。则由之前的分析可知，这种影响必通过进程内通信动作传递出去，即条件中的 t,t' 是一定存在的。

根据性质 2，可以约束定义 6 的条件。

定义 8 给定系统 M，且 M 中进程仅以同步通信的形式交互。对于给定的不干扰策略$\not\rightsquigarrow$，如果对于任意 $u\in D, s,s'\in traces(M), tail(s), tail(s')\in sig(noflow(u))\cup SKIP$，满足，$s\backslash noflow(u)=s'\backslash noflow(u)\wedge tail(s)\neq tail(s')\Rightarrow(M/s)\uparrow\alpha(u)=_f(M/s')\uparrow\alpha(u)$，则称系统 M 满足不干扰策略$\not\rightsquigarrow$。

定义 8 使得只需验证 s,s' 以 $noflow(u)$ 域上不同的通信动作结尾，且除结尾动作以外 s,s' 在 $noflow(u)$ 域上不存在其他分歧的情况。而其它情况根据性质 2 知可以归结到上述情况。

3.3 不干扰策略的自动化验证方法

FDR2 是常用的形式化验证工具，其可接受基于 CSP 语言的描述和验证断言。定义 8 已经采用基于 CSP 的描述形式，但是并不能直接交由形式化验证工具 FDR2 来验证，因为它还不是 FDR2 可验证的断言形式，还需根据 FDR2 可验证断言的要求，构造可证明满足定义 8 的可验证断言。本方案将使用进程提炼检查来验证。在给出验证断言之前，先定义与断言相关的进程和集合如下。

定义 9 给定确定性系统 M，以及不干扰策略$\not\rightsquigarrow$，对于任意 $C\in D$，有如下定义：

进程 $A=\bigcup_{u\in D,u\not\rightsquigarrow C}u$，$B=\bigcup_{u\in D,u\rightsquigarrow C}u$，显然 $A,B\in D$；

进程 A',B',C' 是进程 A,B,C 的拷贝，$A',B',C'\in D$，且 $\forall a\in\alpha(A)$，存在 a 的拷贝 $a'\in\alpha(A')$，且认为 a 和 a' 是不同的动作。B',C' 也有类似的性质。而由 A',B',C' 构成的系统进程记为 M'，M' 是系统 M 的一个拷贝；

$tag(A), tag(B), tag(C), tag(A'), tag(B'), tag(C')$ 分别是 A,B,C,A',B',C' 的字母表的一个拷贝。$\forall a\in\alpha(A)$ 存在 a 的一个标记动作 $ta\in tag(A)$，B,C,A',B',C' 与 A 类似；

标记集 $Tag=tag(A)\cup tag(B)\cup tag(C)\cup tag(A')\cup tag(B')\cup tag(C')$；

不确定进程 $L(M)=M\backslash inner(A)[\alpha(A)]CHAOS(A)$。

同步 M 和 M' 的通信进程：$Com1$，$Com2$

$Com1=(?\ x{:}\alpha(A)->tx->((x'->Com1)\square(?\ y'{:}\alpha(A')-\{x'\}->ty'->Com1')\square(s->Com1'))\square(?\ z{:}\alpha(B)\cup\alpha(C)->z'->tz->Com1)$

$Com1'=(?\ x{:}\alpha(A)\cup\alpha(B)->Com1')\square(?\ z{:}\alpha(C)->tz->tz'->Com1')$

$Com2=(?\ x{:}\alpha(A)->tx->((x'->Com2)\square(?\ y'{:}\alpha(A')-\{x'\}->ty'->Com2')\square(s->Com2'))\square(?\ z{:}\alpha(B)\cup\alpha(C)->z'->tz->Com2)$

$Com2'=(?\ x{:}\alpha(A)\cup\alpha(B)->((?\ y'{:}\alpha(A')\cup\alpha(B')->Com2')\square Com2'))\square(?\ z{:}\alpha(C)->tz->Com2''(z))$

$Com2''(z)=(z'->tz'->Com2')\square(?\ y'{:}\alpha(A')\cup\alpha(B')->Com2''(z))$

复合通信进程：$R1(M,C)$，$R2(M,C)$

$R1(M,C)=L(M)[\alpha(M)]Com1[\alpha(M')]L(M')\uparrow Tag$

$R2(M,C)=L(M)[\alpha(M)]Com2[\alpha(M')]L(M')\uparrow Tag$

根据 $R1$ 和 $R2$ 的定义可知，只有 Tag 内的标记动作才可能出现在 $R1$ 和 $R2$ 中，而 Tag 的标记动作又是根据 $Com1$ 和 $Com2$ 的定义实施的，因此只有在 A 和 A' 产生分歧之前才标记所有的动作，之后只会标记 C 和 C' 的动作。由定义 9 不难看出，$Com2$ 定义了同步规则，如果 A 和 A' 分别在 $L(M)$ 和 $L(M')$ 中选择了不同的执行分支，使得 $Com2'$ 无法将 C 与 C' 达成同步而阻塞，则一定存在 C 或 C' 中一个或多个执行序列在 $R2$ 中无法执行，自然无法由 Tag 集合中的标记所记录。

而 $R1$ 中的 $Com1'$ 并不要求同步 C 与 C'，使得 C 或 C' 中所有序列都可以被执行并记录。很显然，如果 $R1$ 中标记的所有执行序列都在 $R2$ 中出现过，则说明 $R2$ 中 C 与 C' 不会因为 A 和 A' 做出的不同分支选择而无法同步，即 $A \not\succ C$。于是，可以得到下述判定定理。

定理 1 给定系统 M，以及不干扰策略 $\not\succ$，对于任意 $C \in D$，如果 $R1(M,C)$ 和 $R2(M,C)$ 符合定义 9，则系统 M 满足不干扰策略 $\not\succ$ 的充要条件：$failures(R1(M,C)) \subseteq failures(R2(M,C))$。

证明：即证明 $failures(R1(M,C)) \subseteq failures(R2(M,C))$ 与定义 8 的条件等价。

(1)充分性

用反证法：如果存在 $s,s' \in traces(M)$，$s\backslash\alpha(A) = s'\backslash\alpha(A)$，$tail(s)\neq tail(s')$，$tail(s),tail(s')\in sig(noflow(u))\cup SKIP$，有 $(M/s)\uparrow\alpha(C)\neq_f (M/s')\uparrow\alpha(C)$，则 $failures(R1(M,C))\subseteq failures(R2(M,C))$ 不成立。

根据 $Com1$ 和 $Com2$ 的定义可知，在 $L(M')$ 始终同步执行 $L(M)$ 的镜像动作时，$R1$ 和 $R2$ 记录着相同的标记序列。又 M 是确定性进程，$L(M)$ 及其镜像 $L(M')$ 只因 $CHAOS(A)$ 引入了 A 上动作的不确定选择，所以 $L(M)$ 及其镜像的不确定选择只可能在 A 上作出。所以，若令 M 执行 s，M' 执行 s'，则由 $L(M)$ 的定义知，$L(M)$，$L(M')$ 分别执行 $s\backslash inner(A)$ 和 $s'\backslash inner(A)$，记为 t,t'，由性质 1 知，t,t' 只会在最后动作上出现分歧。而由 $Com1$ 和 $Com2$ 的定义可知，当第一次在 A 上发生分歧后，$L(M)$ 和 $L(M')$ 将转向执行 $Com1'$ 和 $Com2'$，而 $Com1'$ 和 $Com2'$ 只在执行到 C 或 C' 中的动作时，才执行相应的 $tag(C)$ 或 $tag(C')$ 中的动作来记录，因此，在执行 s,s' 之后，$R1$ 或 $R2$ 中将出现且仅出现 $tag(C)$ 或 $tag(C')$ 中的动作来反映 C 或 C' 中的所有执行轨迹。不妨设存在 $\beta\in traces(L(M))$，$a1,a2\in\alpha(A)\cup SKIP$，$a1\neq a2$，$t=\beta^\frown a1$，$t'=\beta^\frown a2$，按照 $R1$，$R2$ 的定义，一定存在与 β 对应的标记序列，设为 $t\beta$，显然有 $t\beta^\frown ta1^\frown ta2'\in traces(R1)$，且 $t\beta^\frown ta1^\frown ta2'\in traces(R2)$。

不妨设存在 C 的执行序列 γ，有 $(\gamma,e)\in failtures((M/s)\uparrow\alpha(C))$，$(\gamma,e)\notin failtures((M/s')\uparrow\alpha(C))$，则：

a)当 $\gamma\notin traces((M/s')\uparrow\alpha(C))$ 时，有 $\gamma\in traces((L(M)/t)\uparrow\alpha(C))$，$\gamma\notin traces((L(M)/t')\uparrow\alpha(C))$，不妨设 $\gamma=c1c2\cdots cn$，由 $Com1'$，$Com2'$ 的定义知 γ 的标记序列 $t\gamma=tc1\ tc1'\ tc2\ tc2'\cdots tcn\ tcn'\in traces(R1/(t\beta^\frown ta1^\frown ta2'))$，$t\gamma\notin traces(R2/(t\beta^\frown ta1^\frown ta2'))$，于是有 $failures(R1(M,C))\not\subseteq failures(R2(M,C))$。

b)当 $\gamma\in traces((M/s')\uparrow\alpha(C))$ 时，$L(M)/t$ 与 $L(M')/t'$ 执行 γ 之后在 C 上一定存在不相等的可执行动作集合，不妨设存在 $c\in e$，$\gamma^\frown c\notin traces((L(M)/t')$，$\gamma^\frown c\in traces((L(M)/t)\uparrow\alpha(C))$，同情况 a)可证得 $failures(R1(M,C))\not\subseteq failures(R2(M,C))$。

(2)必要性

用反证法：证明如果存在标记序列 $t\gamma$，有 $(t\gamma,e)\in failures(R1(M,C))$，$(t\gamma,e)\notin failures(R2(M,C))$，则一定存在 $s,s'\in traces(M)$，$s\backslash\alpha(A)=s'\backslash\alpha(A)$，$tail(s)\neq tail(s')$，$tail(s),tail(s')\in sig(noflow(u))\cup SKIP$，有 $(M/s)\uparrow\alpha(C)\neq_f (M/s')\uparrow\alpha(C)$。

根据 $Com1$ 和 $Com2$ 的定义可知，在 $L(M')$ 始终同步执行 $L(M)$ 的镜像动作时，$R1$ 和 $R2$ 记录着相同的标记序列。且由上文的分析知，只有当 $L(M)$，$L(M')$ 在 A 上执行不同的动作之后，$R1$ 和 $R2$ 才会因为分别执行了 $Com1'$ 和 $Com2'$ 而使得执行轨迹不同。所以可以假设 $L(M)$，$L(M')$ 在 A 上执行不同通信动作之前，存在序列 $\beta\in traces(L(M))$，及其标记序列 $t\beta$，有 $t\beta\in traces(R1)$，$t\beta\in traces(R2)$，假设 $L(M)$，$L(M')$ 执行了 β 之后在 A 上执行不同通信动作 $a1,a2\in\alpha(A)\cup SKIP$，$a1\neq a2$，并存在 $t\tau\in traces(R1/(t\beta^\frown ta1^\frown ta2))$ 使得 $t\gamma=t\beta^\frown ta1^\frown ta2^\frown t\tau$，由定义知 $t\tau$ 由 $tag(C)$ 和 $tag(C')$ 的标记动作组成。分两种情况讨论：

a)当 $t\gamma\notin failures(R2(M,C))$ 时，由 $Com1'$ 的定义知 $t\tau=tc1\ tc1'\ tc2\ tc2'\cdots tcn\ tcn'$ 或 $t\tau=tc1\ tc1'\ tc2\ tc2'\cdots tcn$，由 $com1'$ 的定义知一定存在被 $t\tau$ 标记的原始动作序列 $\tau\in traces(L(M)/(\beta^\frown a1)\uparrow\alpha(C))$，且 $\tau=c1c2\cdots cn$，由 $Com2'$ 的定义及假设 $t\gamma\notin failures(R2(M,C))$ 知 $\tau\notin traces(L(M)/(\beta^\frown a2)\uparrow\alpha(C))$，则令 $s=\beta^\frown a1$，$s'=\beta^\frown a2$，显然有 $(M/s)\uparrow\alpha(C)\neq_f (M/s')\uparrow\alpha(C)$。

b)当 $t\gamma\in failures(R2(M,C))$ 时，$R1$ 与 $R2$ 执行 $t\gamma$ 之后存在不相等的可执行动作集合，不妨设存在 c 的标记 $tc\in e$，有 $t\gamma^\frown tc\in traces(R1(M,C))$，$t\gamma^\frown tc\notin traces(R2(M,C))$，同情况 a)得 $\tau^\frown c\in traces(L(M)/(\beta^\frown a1)\uparrow\alpha(C))$，且 $\tau^\frown c\notin traces(L(M)/(\beta^\frown a2)\uparrow\alpha(C))$，同样令 $s=\beta^\frown a1$，$s'=\beta^\frown a2$，有命题成立。

综上所述，$failures(R1(M,C))\subseteq failures(R2(M,C))$ 是系统 M 满足不干扰策略 $\not\succ$ 的充要条件。

4 基于虚拟机和隔离策略的文件系统模型验证

根据定义 7，隔离策略的验证可以通过验证隔离域之间的相互不干扰来达到。本节将通过一个实例演示如何利用上文提出的验证方案来自动化验证实例系统中的隔离策略；将描述和建模一个微内核操作系统中的文件服务器，以及用于监控和保护该文件服务的虚拟机引用监视器。该监视器只有一条策略，就是保护请求文件服务的客户之间的隔离性。将监控器本身与其被监控的服务一同建模，有利于在验证被监控系统的同时，保证监控器自身不会违背监控策略而成为新的安全隐患。因为监控器自身实际上也是一种服务，而且是主要用于隔离用户进程的服务。

4.1 基于虚拟机监控的文件服务模型

根据虚拟机的相关机制[14]，我们可以通过在虚拟机中实现对特定内存访问的捕获和访问检查。为了实现客户进程间的隔离，虚拟机必须能够有效区分服务进程正在为哪个进程服务，以及该服务所使用的内存块或者内存页。并且要能够确保这些内存单元的内容不会被其他请求服务的进程所窃取或修改。具体监控流程如下：如图 3 所示，消息和指令都受到虚拟机的监控。当进程 AP1 请求系统服务时，首先①发送请求 msg 给 Kernel，消息内容包括进程标识 pid；当 kernel 从消息队列中读取消息时，②读操作触发 VM Exit 陷入监控器，虚拟机取得 pid 信息；③通过 VM Entry 虚拟机将控制权交给 Kernel；④Kernel 将消息内容复制给 Service 进程，Service 进程执行某操作 Ins_n；⑤Ins_n 操作试图访问某个数据存储，引发 VM Exit，虚拟机通过 VMCS 获得捕获的指令类型、内存地址等信息，并根据 Policy 判断该操作是否合法，合法则通过⑥模

拟 Ins_n 执行，并返回；否则通过⑦返回到服务进程的失败处理程序。两条虚箭头表明，未通过虚拟机验证的数据访问或消息传递是不能执行的。Intel_VT 技术为内存的访问提供了不依赖于被监控系统的、不可旁路的监控点。

虚拟机为我们提供了有效的监控机制，而要实现共享服务的用户进程相隔离，还须对服务进程的内存操作制定有效的访问策略。虚拟机监控器则可以保证这一策略得以有效的实施。

为了说明监控策略，定义接收文件服务的用户进程集合为 $P=\{P_1,P_2,\cdots,P_n\}$，状态集合 S，及其状态集合上的偏序关系 $<$，如果状态 t 在 s 之后出现，则 $s<t$；且某状态 $s\in S$ 下文件服务为 Pi 服务申请的内存单元为 O_s^i，则文件服务的内存单元集合为 $O=O_s^1\cup O_s^2\cup\cdots\cup O_s^n$；且对于任意单元 $o\in O$，o 的值记为 $val_s(o)$，如果 Pi 在状态 s 下能访问 o，则记 $enable(s,Pi,o)=$ true 否则 false。则满足用户进程隔离性的内存访问策略如下：

$\forall s\in S,i,j\in N,j\neq i\Rightarrow O_s^i\cap O_s^j=\Phi$

$\forall s\in S,o\in O_s^i,j\in N,j\neq i\Rightarrow enable(s,Pj,o)=\text{false}$

$s_1,s_2\in S,o\in O_{s1}^i,o\in O_{s2}^j,i,j\in N,j\neq i\Rightarrow\exists t,s_1<t<s_2,val_t(o)=\text{null}$

第一条策略描述文件服务不被允许同时为两个用户进程申请相同的内存单元来处理请求服务；第二条策略描述文件服务在为一个用户进程服务时，不能读写为另一个用户进程申请的用于处理请求的内存单元；第三条策略描述文件服务在处理完一个进程的请求，并释放为其服务的内存单元之前需先擦除其存储数据，以防止客体重用。

上述策略，是在两个被服务的用户进程完全不需要共享资源的前提下制定的，当然如果两个进程需要共享某些资源，如共享文件 inode 节点。这只需针对具体的共享存储单元放宽隔离策略即可。

如图 3 所示，系统可以简化为下述 4 类实体：用户进程（*AP*）、用户进程（*BP*）、文件服务进程（*FS*）、虚拟监控器（*VMM*）（微内核作为通信媒介，不影响讨论的结果，略去）。文件服务主要维护着与用户进程打开文件相关的控制信息和描述信息。以 *minix*3.0 为例，其数据主要包括以下几类。

（1）与用户进程打开文件相关的数据对象，可用于为多个用户进程存储数据，但是同一时刻只能为一个进程服务。

用户进程打开的文件列表 fl[]。保存所打开文件的文件描述符指针和相关信息；

文件描述符表 filp[]。保存打开文件的 i-node 结点指针和相关信息；

文件 i-node 结点表 inode[]。保存打开文件存放的内存块指针和相关信息；

读入的文件内存块 block[]。

（2）为每一个用户进程单独存储信息的数据对象，如进程控制块信息等。

（3）函数内的临时存储对象，可以给任何用户进程存储服务数据，但是同一时间只能为一个进程服务。

应用进程 AP 或 BP 虽然不能直接访问文件进程（1）－（3）的存储信息。但是，当 AP 或 BP 向 FS 请求服务的过程中，服务处理程序会通过（1）－（3）存储或传递 AP 或 BP 的信息。比如用户进程请求打开、读写或关闭一个文件，必然会更新（1）中某些对象的值，而这些值又会影响返回信息甚至是下一次服务请求的执行。然而，（1）－（3）中哪些数据对象在本次服务中被利用或者被释放，并不是用户进程需要关心的，对它们来说是不可见的，它们只关心这些对象中与自己有关的具体数据的值。

为简单起见，文件服务和监控器的建模不考虑以下情况：

不考虑 AP 和 BP 之间的文件共享和管道。文件共享和管道会使得（1）中的表项可能存在共享，会使得策略描述复杂；而不考虑文件共享和管道也并不会使模型失去一般性，只是避免了为共享信息单独制定策略而已。

不考虑资源耗尽的情况。假设（1）中的节点足够多，不存在无节点可用的情况，避免描述由于争抢最后一个节点所产生的隐蔽通道。

下文首先建立文件服务及虚拟机监控器模型，模型中主要函数、动作的操作语义，以及实体进程的定义都是采用 *CSP* 的描述规范。以便于直接利用定理 1 给出的判定条件通过 *FDR*2 验证用户进程的隔离性。

定义 10 系统 M 除包含定义 1 的所有元素外还包括：

- 进程实体：$AP,BP,FS,VMM\in D$；
- 值集合 V；
- 系统对象实体集 Obj
- 文件进程维护的对象：

控制实体（包括打开文件列表 fl、文件描述符表 $filp$、文件 i-$node$ 表 $inode$）包含指向下级结点的指针对象和存储当前结点信息的存储对象

$Col_Pointer=\{AP_fl_pointer[0..N1],BP_fl_pointer[0..N1],Filp_pointer[0..N2],Inode_pointer[0..N3]\}$

$Col_Inf=\{AP_fl_inf[0..N1],BP_fl_inf[0..N1],Filp_inf[0..N2],Inode_inf[0..N3]$

磁盘块对象 $Block[0..N4]$ 和临时存储对象 $ComParm$；

为用户进程单独保留的存储对象 $ISO_Parm=\{AP_Parm,BP_Parm\}$；

于是，文件系统所有对象的集合为：

$FSO=Col_Inf\cup Col_Pointer\cup\{Block[0..N4],ComParm\}\cup ISO_Parm\subset Obj$。

虚拟监控器维护的策略对象：$VMMO=\{Policy\}\subset Obj$

用户进程 AP 维护的 I/O 对象：$APO=\{AP_in,AP_out\}\subset Obj$

用户进程 BP 维护的 I/O 对象：$BPO=\{BP_in,BP_out\}\subset Obj$

- 取值函数 $val:S\times Obj\rightarrow V$，对象的取值与状态有关；
- 因为 Col_Inf，$Col_Pointer$ 的对象元素是一一对应的，于是有以下函数：

为 Col_Inf 对象取得相对应的指针：$getPt:Obj\rightarrow Obj$

为 $Col_Pointer$ 对象取得相对应的信息：$getInf:Obj\rightarrow Obj$

为保证函数定义的一般性，当输入参数不是 Col_Inf 或 $Col_Pointer$ 时，返回值等于输入值；

- 对象标识集合 $L\subset V$，对象在系统中对应的数值标识，不同对象的标识不同；
- 对象标识函数：$label:FSO\rightarrow L$；

• 对象识别函数：$Object: L \to FSO \cup \{Null\}$。$label$ 的逆函数；

• 对象包含函数：$member: Obj \times 2^{Obj} \to Bool$。$member(o, OE)$ 描述 o 是否在 OE 集合中；

• 取得接收服务的用户进程：$client: A \to D$, if $dom(a)! = FS, client(a) = dom(a)$；

• 对象二元组集 $ObjVecter: S \to 2^{Obj \times V}$。$ObjVecter(s) = \{(o, val(s,o)) \mid o \in Obj, o \neq Policy\}$，为表达简单，记 $ObjVecter_s$ 为状态 s 下二元组集合；记 $valO(ObjVecter_s, obj)$ 为对象 obj 在状态 s 下的取值；

• 策略三元组集 $Policy: S \to 2^{D \times L \times Bool}$。记 $Policy_s$ 为状态 s 下策略；$checkPerm(p, l, Policy_s)$ 为 p 请求的文件服务在状态 s 下是否可访问对象 $Object(l)$，返回类型 $Bool$；

系统 M 动作集合包含以下动作(设 $s, s' = step(s,a)$ 为执行动作 a 前后的状态，$Skip$ 为空操作)：

• FS 读写：$move(p, from, to, ObjVecter_s), p \in D, from, to \in FSO, client(move) = p, dom(move) = FS, enable(s', back(p)) = true$。$move$ 又分解为：抛出异常 $throwEXP$；VMM 调用 $checkRead$ 检查读权限，返回检查结果 $getResult$；FS 读；抛出异常 $throwEXP$；VMM 调用 $checkWrite$ 检查写权限，返回检查结果 $getResult$；修改对象值 $valO(ObjVecter_{s'}, to) = valO(ObjVecter_s.from)$ 或不执行；

• FS 对象清除：$clear(p, o, ObjVecter_s), p \in D, o \in FSO, client(Clear) = p, dom(clear) = FS, enable(s', back(p)) = true$。$clear$ 又分解为：抛出异常 $throwEXP$；VMM 调用 $checkWrite$ 执行下述操作序列：返回取得检查结果 $getResult$；修改对象($valO(ObjVecter_{s'}, o) = 0$ 或不执行；

• 通信：$msg(p, o1, o2, ObjVecter_s), p \in \{AP, BP\}, o1, o2 \in Obj, dom(msg) = p1, \forall a \in A, enable(s', a) = true$。$msg$ 的分解过程与 $move$ 类似，执行效果为：修改对象值 $valO(ObjVecter_{s'}, o2) = valO(ObjVecter_s.o1)$ 或不执行；

• FS 结点链接：$pointerWrite(p, o, v, ObjVecter_s), p \in D, o \in FSO, v \in L, client(pointerWrite) = p, dom(pointerWrite) = FS, enable(s', back(p)) = true$。$pointerWrite$ 又分解为抛出异常 $throwEXP$；VMM 调用 $checkWrite$ 执行下述操作序列：返回取得检查结果 $getResult$；修改对象值 $valO(ObjVecter_{s'}, o) = v$ 或不执行；

• 向虚拟机抛出的异常：$throwEXP!(pc, pid, s, a, o, v, ObjVecter_s), pc = client(a), pid = dom(a)$；

• 读访问控制 $checkRead(pc, pid, o, v, ObjVecter_s, Policy_s), pc, pid \in D, o \in Obj, v \in V, s \in S$。如果 $pid = FS$，判定 $Policy_s$ 是否允许 pid 为 pc 读 $ObjVecter_s$；如果 $pid = pc! = PF$，则需判定 pc 是否有权直接读取 $ObjVecter_s$；

• 写访问控制 $checkWrite(pc, pid, o, v, ObjVecter_s, Policy_s), pc, pid \in D, o \in Obj, v \in V, s \in S$。如果 $pid = FS$，判定 $Policy_s$ 是否允许 pid 为 pc 写 $ObjVecter_s$；如果 $pid = pc! = PF$，则需判定 pc 是否有权直接写 $ObjVecter_s$；

$checkRead$ 和 $checkWrite$ 的执行还依赖下述判定函数：

检查输入数据合法性 $isEvillab: (FSO\text{-}Col) \times V \to Bool$。

检查对象是否为空 $isClear: L \times S \to Bool$。$s$ 在 CSP 中描述为 $ObjVecter$ 在 s 状态下的值。

检查对象是否已被授权：$isAuth: L \times S \to Bool$。$s$ 在 CSP 中描述为 $Policy$ 在 s 状态下的值。

策略修改动作：$setPolicy(p, l, bool, Policy_s), p \in D, l \in L, bool \in Boolean, dom(setPolicy) = VMM$，执行后有：

$checkPerm(p, l, Policy_{s'}) = bool$

• 由虚拟机返回检查结果：$getResult!(b), b \in Bool$；

• AP, BP 服务请求动作：$applyService(p), p \in D, dom(applyService(p)) = p, \forall s \in S, a \in A, dom(a) = FS, client(a) = p \Rightarrow Enable(step(s, applyService(p)), a) = true$；

• AP, BP 读动作：$showOut(p)! \ v, p \in D, v \in V$，读出 AP_out 或者 BP_out 赋值；

• 文件服务请求动作：$applyFS!(p, ObjVecter_s)$，文件系统接收到请求后，可执行任意文件操作，即 $\forall a \in A, dom(a) = FS \Rightarrow enable(s,a)$，$s$ 为执行 $applyFS$ 后状态；

• 初始状态：$\forall a \in A, dom(a) \in \{AP, BP\} \Rightarrow Enable(s_0, a) = true$；$\forall o \in \{Fproc_AP_fl[1..N1], AP_Parm\} \Rightarrow checkperm(AP, label(o), Policy_{s0}) = true$；$\forall o \in \{Fproc_BP_fl[1..N1], BP_Parm\} \Rightarrow checkperm(BP, label(o), Policy_{s0}) = true$；

• AP, BP 的进程描述

$pAP(ObjVecter_s) = (getto?\ o -> msg(AP, AP_in, o, ObjVecter_s))$

$[](getfrom?\ o -> msg(AP, o, AP_out, ObjVecter_s))$

$[](showOut!\ val(AP_out) -> pBP(ObjVecter_s))$

$[]applyFS!\ AP!\ ObjVecter_s$

$pBP(ObjVecter_s) = (getto?\ o -> msg(BP, BP_in, o, ObjVecter_s))$

$[](getfrom?\ o -> msg(BP, o, BP_out, ObjVecter_s))$

$[](showOut!\ val(BP_out) -> pBP(ObjVecter_s))$

$[]\ applyFS!\ BP!\ ObjVecter_s$

• FS 进程描述

$pFS(p) = applyFS?\ pid?\ ObjVecter_s$

$->(getFromandTo?\ o1?\ o2!\ ObjVecter_s$

$-> move(p, o1, o2, ObjVecter_s))[](getClearObj?\ o$

$-> clear(p, o, ObjVecter_s))[](getObjandLabel?\ v$

$-> pointerWrite(p, o, v, ObjVecter_s))$

• VMM 进程描述

$pVMM(Policy_s) =$

$(throwEXP?\ pc?\ pid?\ a?\ o?\ v?\ ObjVecter_s ->$ if$(a = read)$ then

$checkRead(pc, pid, a, o, v, ObjVecter_s, Policy_s)$

$else\ checkWrite(pc, pid, a, o, v, ObjVecter_s, Policy_s)$

系统 M 进程描述

$System = (pAP \mid\mid\mid pBP)[\mid aPF \mid]pFS[\mid aPFV \mid]pVMM$

式中，$aPF = \{\mid applyFS \mid\}$；$aPFV = \{\mid throwEXP, getResult \mid\}$

由定义 10 知，初始状态，虚拟监控器中的 $policy$ 被赋予了初始权限，随后，自动机由 AP 或者 BP 的服务请求动作启动。服务请求动作又可以启动服务进程相关动作为其服务。AP 或者 BP 在与 FS 交互过程中 AP_in 或者 BP_in 为 FS 执行提供参数。同时，FS 执行结果又可以通过 AP_out 或者 BP_out 返回给服务请求者。

VMM 主要负责访问控制和策略的读写，由 VMM 动作的执行描述知其访问控制策略思想包括：(1)在 $policy$ 中关

联用户进程和 *FS* 为该进程服务时所使用的对象;(2)*FS* 为某用户进程服务时,不能访问与其他用户进程关联的对象;(3)当 *FS* 要启用某对象为用户进程服务时,必须保证该对象不与其他用户进程关联且值为空;(4)当 *FS* 要在服务中释放某对象之前,必须清空该对象;(5)指针信息对象集 *Col_Pointer* 中对象只能赋 L 集合中的标识信息;(6)用户进程只能通过消息与 *FS* 维护的 *AP_Parm* 或者 *BP_Parm* 交互。因此(1),(2)保证了同一对象不会同时为不同用户进程服务;(3),(4)避免了客体重用;(5)限定了指针的取值范围;(6)限定了进程间通信的存储客体。另一方面,用户进程仅关心模型中为自己服务的对象的值,而选择哪些对象为哪个进程服务是文件系统内部的操作,与请求服务的进程无关。因此,在上述模型中对 *Col_Pointer* 集合内对象的地址赋值是 *FS* 内部选择的,对用户进程透明,也不会存储和传递对用户进程有意义的信息。因此,$AP \perp BP$ 等价于验证:进程 *AP* 以任意次序执行任意动作,或向服务进程传输任意 *AP_in*,都不能影响 *BP* 的执行及其输出对象 *BP_out* 的值,反之亦然。

定义 10 中对函数、动作、进程的描述已是基于 CSP 的描述,只需增加部分对象定义和少数函数的描述就可以作为验证代码,由于篇幅所限,不再详细给出。基于验证代码,可以分别假定 *AP* 或 *BP* 为干扰源或被干扰进程,而 *FS*[]*VMM* 显然对应着定义 9 中的进程集合 *B*,因此,可以分别以 *AP* 或 *BP* 之一为干扰源,另一个为被干扰进程,两次根据定义 9 计算与定义 10 中系统 *M* 相关的 *System'*,*Com1*,*Com2*,*Com1'*,*Com2'*,*R1*,*R2*,并通过在 FDR2 中验证断言 *Assert R2*[*F*=*R1* 来验证隔离关系。由于篇幅限制,具体的过程不再一一给出。两次断言验证的结果都如图 4 所示,显然,由定理 1 知,*AP* 和 *BP* 在定义 10 描述的系统中是相互隔离的。

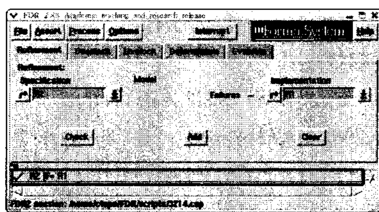

图 4 FDR2 验证结果截图

5 相关工作

本文给出的隔离语义主要基于 Haigh and Young[2] 在 1986 年提出的非传递不干扰模型,随后 Rushby[6,9] 又利用进程代数理论对其进行了重新阐述和修正。但是其讨论的信息流策略都是主要集中在信息域间直接的信息交互上,而实际上,如上文所述,对进程间利用公共第三方的间接交互策略的讨论同样重要,并具有广泛的应用背景。Rushby 的隔离性证明理论[9]也提出了对多个共享系统资源的用户进行隔离验证的方法,但是该方法主要从用户的输入输出数据入手分析,并没有考虑数据对用户行为的影响。另一方面,Rushby 在文献[9]中主要给出了信息隔离的判定条件,并证明了其正确性,但对如何实施验证并没有给出相关的说明,这对于一个复杂的系统验证而言缺乏可操作性。

使用进程代数 CSP[7] 研究信息流理论于 1987 年由 Foley[11] 最先提出。1990 年,Ryan 使用 CSP 对无干扰模型进行了全新表述[12]。此后,进程代数被作为研究信息流安全性质的合适工具之一。Ryan[13],Roscoe[5] 等人也在 CSP 框架下对信息流安全性质进行研究。本文也正是在上述研究的基础上,借助 CSP 对通信进程的描述和分析能力来对一个实际的系统进行模型描述和验证,注重方案的可操作性。

结束语 本文基于不干扰理论分析了系统中进程或模块隔离所具有的性质,通过定义新的不干扰关系以及系统满足该不干扰关系所应满足的条件,准确定义了隔离的语义。并借助通信顺序进程 CSP 给出了描述、分析和验证一个系统是否能够满足给定的隔离策略的方法,验证方案可借助形式化验证工具 FDR2 实现自动化。本文作者下一步的工作将利用该理论及其验证方案描述和验证操作系统中不同服务进程、内核线程之间的隔离关系,并在进一步放宽隔离策略的基础上研究信道策略的形式化描述和验证方法。

参 考 文 献

[1] Goguen J, Meseguer J. Security policies and security models[C]// Proceedings of 1982 IEEE Symposium on Research in Security and Privacy. Los Alamitos: IEEE Computer Society Press, 1982: 11-20

[2] Haigh J, Young W. Extending the non-interference model of MLS for SAT[C]// Proceedings of 1986 Symposium on Security and Privacy. Oakland, CA: IEEE Computer Society, 1986: 232-239

[3] Roscoe AW, Woodcock J C P, Wulf L. Non-interference through determinism[J]. Journal of Computer Security, 1996, 4(1): 27-54

[4] Roscoe A W. CSP and determinism in security modeling[C]// Proceedings of 1995 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society Press, 1995: 114-127

[5] Roscoe A W, Goldsmith M H. What is Intransitive Noninterference? [C]// Proceedings of the 12th Computer Security Foundations Workshop. Mordano, Italy: IEEE Computer Society Press, 1999: 228-238

[6] Rushby J. Noninterference, transitivity, and channel-control security policies[R]. Menlo Park: Stanford Research Institute, 1992

[7] Hoare C A R. Communicating sequential processes[R]. Prentice-Hall, 1985

[8] Formal Systems (Europe) Ltd. FDR2 User Manual[OL]. http://www.fsel.com/documentation/fdr2/html/index.html

[9] Rushby J. Formal verification of the unwinding theorem for intransitive noninterference security policies[R]. Menlo Park: Computer Science Laboratory, 1991

[10] Rushby J. Proof of Separability—a verification technique for a class of security kernels[C]// Proceedings of 5th International Symposium on Programming. Turin, 1982: 352-367

[11] Foley S N. A universal theory of information flow[C]// IEEE Symposium on Security and Privacy. Oakland, 1987: 116-121

[12] Ryan P Y A. A CSP formulation of noninterference and unwinding[C]// Cipher: IEEE Computer Society Technical Committee Newsletter on Security & Privacy. Washington: IEEE Computer Society Press, 1991: 19-27

[13] Ryan P Y A, Schneider S A. Process algebra and non - interference[J]. Journal of Computer Security, 2001, 9(1/2): 75-103