マルチメディア通信と分散処理 86－9
グループウェア 26－9
（1998. 1. 29）

Information Flow Control in Distributed Systems

Masashi Yasuda, Takayuki Tachikawa, and Makoto Takizawa

Tokyo Denki University
E-mail {masa, tachi, taki}@takilab.k.dendai.ac.jp

Distributed applications are realized by cooperation of multiple objects. Each object is manipulated through an operation supported by the object and then the operation may further invoke operations of other objects, i.e. nested operations. Purpose-oriented access rules indicate what operation in each object can invoke operations of other objects. Information flow among the objects occurs if the requests and responses of the operations carry data. Only the purpose-oriented access rules which imply legal information flow are allowed. In this paper, we discuss how to specify the access rules so that the information flow occurring in the nested invocation of the operations is legal.

分散システムにおける情報流制御

安田 昌史　立川 敬行　滝沢 誠

東京電機大学理工学部経営工学科

分散応用は、複数のオブジェクトからなるグループが通信網を通じてメッセージの交換を行うことで実現される。オブジェクトはデータ構造と抽象型操作演算の対により表現され、オブジェクトにより提供される操作演算を通してのみ操作することができる。また、操作演算はさらに他のオブジェクトが提供する操作演算を呼び出すことがある。これを入れ子型演算という。複数のオブジェクトが協調動作する環境下では、オブジェクト間における不正な情報流を防止することが重要である。本論文では、目的指向のアクセス制御モデルにおける正しいアクセス規則の定義と不正な情報流の解析方法について論じる。

1 Introduction

In client-server systems, the application programs in the clients manipulate the resources in the servers. Units of the resources like databases are named *objects*. It is significant to consider what subject s can access what object o by what operation t in the access control model. An *access rule* is given in a tuple $\langle s, o, t \rangle$ [6]. The system is *secure* if and only if (*iff*) every object is accessed only according to the access rules. However, the access control model cannot resolve the containment problem [6] where the information illegally flows among subjects and objects. The lattice-based model [1,5] aims at protecting against illegal information flow among the entities. One security class is given to each entity in the system. A *flow* relation among the security classes is defined to denote that information in one class s_1 can *flow into* s_2. In the *mandatory* model [1,9], the access rule $\langle s, o, t \rangle$ is specified so that the flow relation between the subject s and the object o holds. For example, s can read o only if the security class of o *can flow* to the class of s. Here, only *read* and *write* are considered as access types of the objects. In the role-based model [10,13], a *role* is defined in a set of operations on objects. The role represents a function or job in the application. The access rule is defined to bind a subject to the roles.

Distributed applications are modeled in an object-based model like CORBA [7]. Here, the system is a collection of objects. Each object is an encapsulation of more abstract data structure and operations than *read* and *write*. The objects are manipulated only through operations supported by themselves. The access rules are defined based on the operation types. For example, a person s may withdraw money from a bank o in order to do house-keeping. However, s cannot get money from o to go drinking. Thus, it is essential to discuss the *purpose* of s to access o by t. The *purpose-oriented* model [12] is proposed where an access rule shows for what each subject s manipulates an object o by an operation t of o so as to keep the information flow legal. The *purpose* of s to access o by t is modeled to be what operation u of s invokes t to manipulate o. That is, the purpose-oriented access rule is specified in a form $\langle s : u,\ o : t \rangle$. In the object-based system, on receipt of a request *op* from an object o_1, an object o_2 computes *op* and then sends back the response of *op* to o_1. Here, if the request and the response carry data, the data in o_1 and o_2 is exchanged among o_1 and o_2. Furthermore, the operations are nested in the object-based system. Even if each purpose-oriented rule between a pair of objects satisfies the information flow relation, some data in one object may illegally flow to another object through the nested invocation of operations. In this paper, we discuss what the information flow is legal in the nested invocations in the purpose-oriented model of the object-based system.

In section 2, we present the purpose-oriented model in the object-based system. In section 3, we discuss the legal information flow in the purpose-oriented model.

2 Purpose-Oriented Models

2.1 Mandatory model

An access rule $\langle s, o, t \rangle$ means that a subject s can manipulate an object o by an operation type t [6]. The basic model implies a *containment* prob-

lem, where illegal information flow occurs. The lattice-based model [1, 5] is proposed to keep the information flow legal in the system. Here, one *security class* is given to each entity. For each entity e_i in E, let $\lambda(e_i)$ denote a security class given to e_i.

The legal information flow is denoted by the *can-flow* relation "$\rightarrow$" [1, 5]. A security class s_1 and s_2, s_1 *can flow* to s_2 ($s_1 \rightarrow s_2$) *iff* the information in an entity of s_1 can flow into an entity of s_2. s_1 and s_2 are *equivalent* ($s_1 \equiv s_2$) *iff* $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_1$. $\rightarrow$ is reflexive and transitive.

[Definition] For every pair of security classes s_1 and s_2 in S, $s_1 \prec s_2$ *iff* $s_1 \rightarrow s_2$ but $s_2 \not\rightarrow s_1$.□

Here, s_2 *dominates* s_1 ($s_1 \preceq s_2$) *iff* $s_1 \prec s_2$ or $s_1 \equiv s_2$. $s_1 \preceq s_2$ means that s_2 is more sensitive than s_1. For example, suppose there are primitives of information: Network(N) and Database(D). There are classes $S = \{\phi, \{N\}, \{D\}, \{N, D\}\}$. Here, $\alpha \preceq \beta$ if $\alpha \subseteq \beta$. For example, $\{N\} \preceq \{N, D\}$. $\langle S, \preceq, \cup, \cap \rangle$ is a lattice where $\cup$ and $\cap$ are the least upper bound (*lub*) and the greatest lower bound (*glb*), respectively. For example, information in a subject s can flow to both object o_1 and o_2 if $\lambda(s) \preceq \lambda(o_1) \cap \lambda(o_2)$.

In the mandatory model [1, 9], the access rule is defined so as to satisfy "$\preceq$". We have to decide if a subject s can manipulate an object o by an operation t. There are three types of operations, i.e. $T = \{read, write, modify\}$. If s reads o, the information in o is derived by s, i.e. information in o flows to s. Hence, $\lambda(s) \succeq \lambda(o)$ is required to hold. If s writes o, the data of s is stored in o, i.e. information in s flows to o. Hence, $\lambda(s) \preceq \lambda(o)$. Lastly, the s's modification of o means that s reads data from o and then writes the result obtained from the data into o. Hence, $\lambda(s) \preceq \lambda(o)$ and $\lambda(s) \succeq \lambda(o)$, i.e. $\lambda(s) \equiv \lambda(o)$.

2.2 Object-based model

The object-based system is composed of multiple objects. Each object o_i supports more abstract data structure and operations than read and write considered in the mandatory model. In addition, o_i is *encapsulated* so that o_i can be accessed only through the operations supported by o_i.

Suppose an object s invokes op_i to manipulate another object o_i. First, we assume that all operations in the system are unnested. s sends a request message q of op_i to o_i. On receipt of q, o_i computes op_i and sends the response r back to s. q carries the input of op_i and r carries back the output of op_i. In addition, op_i may change the state of o_i by using the input. Here, the information in s may flow into o_i if q carries some data in s. op_i may derive the data from o_i and then return the data to s. Here, the information in o_i may flow out to s if r carries the data derived by op_i from o_i. Thus, it is significant to make clear the input and output of op_i to clarify the information flow relation between s and o_i. Each operation op_i of o_i is characterized by (1) input data (I_i), (2) output data (O_i), and (3) state transition of o_i. The input data I_i exists if some data flows from s to o_i. For example, I_i exists if the request of op_i includes the data. The output data O_i exists if some data in o_i flows out to s. For example, O_i is the data carried by the response of op_i. In this paper, we make the following assumptions:

[Assumptions]

(1) The communication among the objects is secure, e.g. messages are encrypted [6].

(2) Each operation op_i of o_i is reliable, i.e. op_i does not malfunction. □

Only data stored in o_i can flow out from o_i to s and the data in s can flow to o_i in the computation of the operation of o_i.

The operations op_i of the objects are classified into the following *flow* types $\tau(op_i)$ from the information flow point of view [Figure 1]: *non-flow* (NF), *flow-in* (FI), *flow-out* (FO), and *flow-in/out* (FIO). In the NF operation op_i, there is no information flow from or to o_i. In addition, op_i does not change o_i. Even if the input data I_i exists, no information in s flows to o_i unless op_i changes o_i. Similarly, no information in o_i flows out to s unless the output data O_i is derived from o_i. If op_i is internally realized by read or write, op_i reads data from o_i or changes o_i. In addition, some data may be brought to op_i from s. However, unless the data is brought between s and o_i, there is no information flow between s and o.

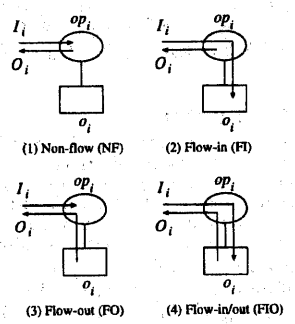

Figure 1: Information flow.

The FI operation op_i changes o_i by using I_i which includes information in s. Hence, the information in s may flow into o_i. *write* is an example of the FI. In addition, o_i is updated without I_i. For example, a *count-up* operation does not have the input but changes the *counter*.

The FO operation op_i does not change o_i. The output O_i of op_i carries the information in o_i back to s. Here, the information in o_i may flow to s. *read* is FO on the file.

The FIO operation op_i changes o_i by using I_i and sends O_i including the information in o_i back to s. Not only the information in s may flow into o_i but also the information in o_i may flow out to s. In *modify*, s first reads O_i in o_i and writes to o_i. FIO may not carry I_i like FI.

The subject s is allowed to manipulate o_i by an operation op_i of o_i according to the following rules.

[Extended access rules]

(1) $\tau(op_i) \in$ {NF, FI} only if $\lambda(s) \preceq \lambda(o_i)$.

(2) $\tau(op_i) \in$ {NF, FO} only if $\lambda(s) \succeq \lambda(o_i)$.

(3) $\tau(op_i) \in$ {NF, FI, FO, FIO} only if $\lambda(s) \equiv \lambda(o_i)$.□

The types of operations and the security class $\lambda(o_i)$ of the object o_i are specified when o_i is defined based on the semantics of o_i. Each time s accesses o_i by op_i, op_i is accepted if $\tau(op_i)$ and $\lambda(o_i)$ satisfy the access rules. If satisfied, op_i is allowed to be computed in o_i.

[Example 1] We consider an example of a World Wide Web (*WWW*) [2] object w accessed from two hosts h_1 and h_2. Here, w is an abstraction of the *httpd* server's service supporting *GET* and *POST* methods. *GET* is an FO type operation because the output data is derived from w. *POST* is FI because w is changed by using the input data. If $\lambda(h_1) \preceq \lambda(w)$ and $\lambda(h_2) \succeq \lambda(w)$, h_1 can *POST* but cannot *GET* data in w, and h_2 can *GET* but cannot *POST* data in w. w can also support abstract operations like Common Gateway Interface (CGI). In the CGI, the users can define operations in the access configuration file such as *.htaccess* to manipulate the page objects. Here, *.htaccess* includes the following :

⟨Limit *GET*⟩, allow from h_2, deny from h_1, ⟨/Limit⟩.

⟨Limit *POST*⟩, allow from h_1, deny from h_2, ⟨/Limit⟩. □

2.3 Purpose-oriented model

First, we define secure objects.

[Definition] An object o_i is *secure* iff

(1) o_i can be only accessed through the operations supported by o_i,

(2) no operation of o_i malfunctions, and

(3) a pair of operations op_1 and op_2 can exchange data only through the state of o_i. □

If data d flowing from an object o_i to another o_j is neither derived from o_i nor stored in o_j, it is meaningless to consider the information flow from o_i to o_j. If data derived from o_i is stored in o_j, the data may flow out to other objects. We assume that every object is secure.

In the access control model, an access rule $\langle s, o_i, op_i \rangle$ means that a subject s manipulates an object o_i through an operation op_i. Suppose a person p accesses a bank account object b of p. p can withdraw money from b if p uses the money to do the house-keeping. However, p cannot get money from b to go drinking. In order to make the system secure, it is critical to consider a *purpose* for which s manipulates o_i by t_i in addition to discussing whether s can manipulate o_i by t_i. Suppose o_i manipulates o_{ij} by invoking an operation op_{ij} of o_{ij}. Here, the *purpose* of o_i for manipulating o_{ij} is modeled to show which *operation* in o_i invokes op_{ij} of o_{ij}. Hence, the access rule is written in a form $\langle o_i : op_i,\ o_{ij} : op_{ij} \rangle$ in the purpose-oriented model while $\langle o_i, o_{ij}, op_{ij} \rangle$ is specified in the mandatory model. op_i shows the *purpose* for which o_i manipulates o_{ij} by op_{ij}. Here, o_i and o_{ij} are named *parent* and *child* objects of the access rule, respectively.

[Purpose-oriented (PO) rule] The access rule $\langle o_i : op_i,\ o_{ij} : op_{ij} \rangle$ means that o_i can manipulate o_{ij} through an operation op_{ij} invoked by op_i of o_i. □

[Example 2] Suppose a person object p can withdraw money from a bank account b of p [Figure 2]. This is shown in an access rule $\langle p : housekeep,\ b : withdraw \rangle$. However, an access rule $\langle p, b, withdraw \rangle$ in the access control model only shows that p can *withdraw* money from b. □

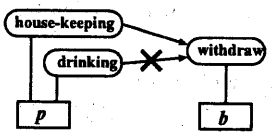

Figure 2: Purpose-oriented access control.

3 Information Flow

We discuss what purpose-oriented rules are allowed to be specified from the information flow point of view.

3.1 Computation model

Each object o computes an operation op on receipt of a request op. o creates a thread of op named an *instance* of op. op may invoke operations $op_1,\ldots,op_l$ where each op_i is computed on an object o_i. There are *synchronous* and *asynchronous* ways for op to invoke op_i. In the synchronous invocation, op waits for the completion of op_i. In the asynchronous one, op does not wait for the completion of op_i, i.e. op_i is computed independently of op. Furthermore, there are *serial* and *parallel* invocations. In the serial invocation, op serially invokes $op_1,\ldots,op_l$, i.e. op invokes op_i after the completion of op_{i-1}. Hence, the information carried by the response of op_{i-1} may flow to op_i. On the other hand, op invokes $op_1, \ldots, op_l$ in parallel. Each op_i is computed on o_i independently of another op_j. This means that the information carried by the response of op_i does not flow to op_j while flowing to op.

In this paper, we consider the serial and parallel synchronous invocations. The invocations of $op_1,\ldots,op_l$ by op are represented in an ordered *invocation tree*. In the invocation tree, each branch $(op \rightarrow op_i)$ shows that op invokes op_i. In addition, $op_1,\ldots,op_l$ are partially ordered. If op_i is invoked before op_j, op_i precedes op_j $(op_i \rightarrow op_j)$. For example, suppose a user serially invokes two operations op_1 and op_2. op_1 invokes op_{12} and op_{13} in parallel after op_{11}. This is represented in the invocation tree as shown in Figure 3. "$\rightarrow$" shows the computation order of the operations. We assume that no operation instance appears multiple times in the tree.

3.2 Nested invocation

In the object-based system, the operations are invoked in the nested manner. Suppose an object o invokes an operation op_i in o_i. op_i further invokes operations $op_{i1},\ldots,op_{il_i}$ where each op_{ij} is in o_{ij}. op_i in o_i communicates with o and o_{ij} while exchanging data with o. Hence, op_i is modeled to

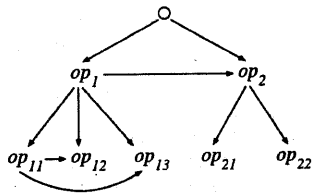

Figure 3: Invocation tree.

be a collection of inputs α_1, α_2, and α_3, and outputs β_1, β_2, and β_3 as shown in Figure 4. Here, α_1 means the input data I_i from o to op_i. For example, the request of op_i carries the input data as α_1. β_1 means the output data to o. The response of op_i is an example of β_1. β_2 shows that o_i is updated by using data carried by β_2. For example, the data of β_2 is stored in o_i. α_2 means that the information derived from o_i is stored in op_i. β_3 means that some data is output to o_{ij}. For example, the request of another operation op_{ij} with the input I_{ij} is sent to o_{ij}. α_3 shows that some output is carried to o_i from o_{ij}. For example, the response with the output data is sent from op_{ij}.

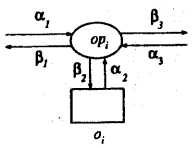

Figure 4: Input and output.

3.3 Invocation graph

An *invocation graph* is introduced to show the information flow relation among operations. Each node indicates an operation. There are *request* (Q) and *response* (S) edges. If an operation op_i of an object o_i invokes op_j of o_j, there is a Q edge from op_i to op_j denoted by a straight arrow line, i.e. a connection between β_3 of op_i and α_1 of op_j. There are the following points to be discussed on the Q edge ;

(1) whether or not op_i sends data in o_i to op_j, and
(2) whether or not op_j changes the state of o_j.

Hence, there are four types of Q edges as shown in Figure 5. The first type (1) is named QNN. op_i sends a request message op_j without data to o_j and op_j does not change o_j. That is, β_3 of op_i and α_1 of op_j do not carry data. In addition, β_2 of op_j does not carry data. There is no information flow from o_i to o_j. The second (2) is QON. op_i sends a request op_j with data to o_j but op_j does not change o_j. Although some data is derived from o_i, the data does not flow to o_j. The third (3) is QNI. op_j changes o_j while op_i does not send data to o_j. Some data flows into o_j but the data does not flow out from o_i. The last (4) is named QOI. Here, op_i sends data to o_j and op_j changes o_j. Some data in o_i flows to o_j. α_2 and β_3 of op_i and α_1 and β_2 of op_j carry data.

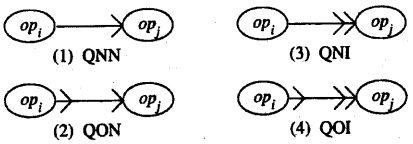

Figure 5: Request (Q) edges.

Next, let us consider the response (S) edges which show information flow carried by the responses from o_j to o_i. The S edges are indicated by dotted arrow line. There are the following points to be discussed on the S edges ;

(1) whether or not op_j sends data in o_j to op_i, and
(2) whether or not op_i changes the state of o_i.

Here, there are four types of S edges as shown in Figure 6. The first type (1) is referred to as SNN, where no information flow from o_j to o_i. The second (2) is SNO, where op_j sends o_i the response with data derived from o_j, but op_j does not change o_i. The third (3) is SIN. op_i changes o_i but op_j sends the response without data to o_i. The fourth (4) is SIO. Here, op_j sends back the response with data derived from o_j to o_i and op_i changes o_i. That is, data in o_j flows to o_i.

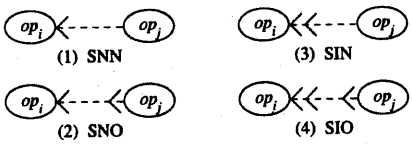

Figure 6: Response (S) edges.

If op_i invokes op_j, a *couple* of Q and S edges exist. There are sixteen possible couples for each invocation. One couple is denoted in a form α/β, where $\alpha \in$ {QNN, QNI, QON, QOI} and $\beta \in$ {SNN, SNO, SIN, SIO}.

3.4 Flow graph

The nested invocation is represented in an *invocation* tree as presented in the previous subsection. Here, suppose that an operation op_i invokes op_j in an invocation tree T. There are a Q edge Q_{ij} from the parent op_i to the child op_j and an S edge S_{ij} from op_j to op_i. Thus, each branch between op_i and op_j represents a couple of Q_{ij} and S_{ij} edges between op_i and op_j. Here, let *root* (T) denote a root of the tree T. In order to analyze the information flow among the operations, a *flow graph* F is obtained from the invocation tree T by the following procedure.

[Construction of flow graph]

(1) Each node in F indicates an operation of T.
(2) For each node op_d connected to the parent by QNI or QOI edge in T, a path P from *root*

(T) to op_d is obtained. For each node op_s in P, there is a directed edge $op_s \rightarrow op_d$ in F if there is a QON or QOI edge from op_s to a child node in P [Figure 7 (1)].

(3) For each node op_p in T, $op_{c_1} \rightarrow op_{c_2}$ if op_{c_1} and op_{c_2} are descendents of op_p in T, which are included in different subtrees of op_p, op_{c_1} has an SNO or SIO edge with the parent of op_{c_1}, and op_{c_2} has a QNI or QOI edge with the parent of op_{c_2} and op_{c_1} precedes op_{c_2} in T [Figure 7 (2)].

(4) $op_1 \rightarrow op_3$ if $op_1 \rightarrow op_2 \rightarrow op_3$ [Figure 7 (3)].

Let us consider a leaf node op_l in the invocation tree T. A leaf node does not invoke other operations. If op_l is invoked with some data and sends back the response, op_l may forward the input data carried by the request to the parent of op_l. Therefore, we have to consider the following additional rules for each leaf node op_l.

(5) For each node op_l connected to the parent by an SNO or SIO edge in T, a path P from *root* (T) to op_l is obtained. For each node op_d in P, there is a directed edge $op_l \rightarrow op_d$ in F if there is an SIN or SIO edge from a child node to op_d [Figure 7 (4)].

(6) For each leaf node op_l, a path P from *root* (T) to op_l is obtained. For every node op_s in P, $op_s \dashrightarrow op_l$ if op_s is connected with the child in a QON or QOI edge. For each node op_d in P, there is a directed edge $op_l \dashrightarrow op_d$ in F if op_d is connected to the child in an SIN or SIO edge. For each node op_s in P, there is a directed edge $op_s \rightarrow op_d$ if (1) $op_s \dashrightarrow op_l$ or $op_s \rightarrow op_l$ and (2) $op_l \dashrightarrow op_d$ [Figure 7 (5)].

(7) For each node op_i which is connected to the parent in SNO or SIO edge, a path P from root (T) to op_i is obtained. If op_j in P is connected to the child in QNI or QOI and SIO or SIN edge, $op_i \rightarrow op_j$ [Figure 7 (6)]. □

By using the rules, a flow graph F is obtained from the invocation tree T. A directed edge $op_i \rightarrow op_j$ in F denotes that there is information flow from op_i to op_j.

[Example 3] Let us consider an example of the SSI command [2] [Figure 8]. A browser B accesses an httpd server to GET a page P including in the same server. Suppose B invokes *GET* on P. P includes two files F_1 and F_2. After including these pages, B caches these data to a disk D. Figure 9 (1) and (2) show the invocation tree T and the flow graph F obtained from T, respectively. According to the flow graph F, we find that the data in the files F_1 and F_2 can flow to the disk D. □

3.5 Access rules

The flow graph shows the possible information flow to occur if the operations are invoked according to the purpose-oriented rules. Each purpose-oriented access rule $\langle o_i : op_i, o_j : op_j \rangle$ is allowed to be specified if the rule satisfies the information flow relation among the objects. The directed edge $\rightarrow$ between op_i and op_j is *legal* in F if the following rule is satisfied. If so, op_i and op_j are referred to as *legally* related.

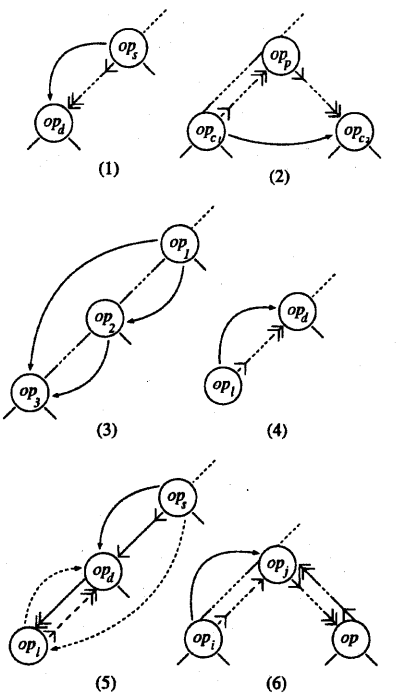

Figure 7: Directed edges.

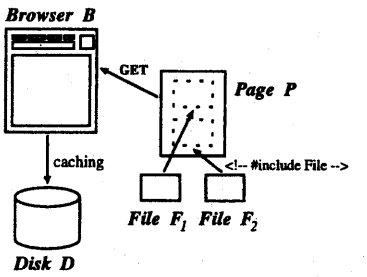

Figure 8: SSI–"include."

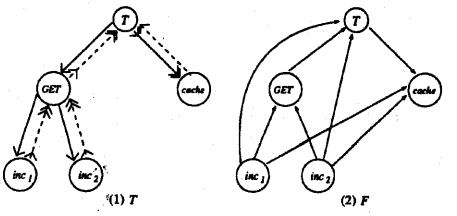

Figure 9: Flow graph.

[Flow rules]
(1) $op_i \rightarrow op_j$ only if $\lambda(o_i) \preceq \lambda(o_j)$.
(2) $op_i \leftarrow op_j$ only if $\lambda(o_i) \succeq \lambda(o_j)$.
(3) $op_i \leftrightarrow op_j$ only if $\lambda(o_i) \equiv \lambda(o_j)$.

Even if an access rule $\langle o_j : op_j, o_k : op_k \rangle$ is specified, op_i cannot invoke op_j if op_j and op_k are not legally related to the information flow relation. Here, $\langle o_i : op_i, o_j : op_j \rangle$ is allowed to be specified if all the directed edges incident to and from op_i and op_j are legal.

[Example 4] In the flow graph shown in Figure 9, $GET \rightarrow open$ is legal only if $\lambda(P) \preceq \lambda(B)$, and $inc_1 \rightarrow GET$ is legal only if $\lambda(F_1) \preceq \lambda(P)$. That is, the PO rules $\langle B : open, P : GET \rangle$ and $\langle P : GET, F_1 : inc_1 \rangle$ are legal. However, $inc_2 \rightarrow GET$ is illegal if $\lambda(F_2) \succeq \lambda(P)$. That is, the rule $\langle P : GET, F_2 : inc_2 \rangle$ is illegal. Then, illegal information flow between P and F_2 may occur if GET on P invokes inc_2 on F_2. Hence, the PO rule $\langle B : open, P : GET \rangle$ is illegal and B cannot invoke GET on a page P through $open$. □

4 Concluding Remarks

In the distributed systems, objects support more abstract operations than *read* and *write*. In the *purpose-oriented* access control model [12], it is discussed why an object manipulates other objects while the mandatory model discusses if each subject can access an object by an operation. In addition, the operations of the objects are nested. The access rules have to satisfy the information flow relation among objects. In this paper, we have discussed how to validate the purpose-oriented access rules.

References

[1] Bell, D. E. and LaPadula, L. J., "Secure Computer Systems: Mathematical Foundations and Model," *Mitre Corp. Report* No. M74-244, *Bedford, Mass.*, 1975.

[2] Berners-Lee, T., Fielding, R., and Frystyk, H., "Hypertext Transfer Protocol – HTTP/1.0," *RFC-1945*, 1996.

[3] Bertino, E., Samarati, P., and Jajodia, S., "High Assurance Discretionary Access Control in Object Bases," *Proc. of the 1st ACM . on Computers and Communication Security*, 1993, pp. 140–150.

[4] Castano, S., Fugini, M., Matella, G., and Samarati, P., *Database Security*, Addison-Wesley, 1995.

[5] Denning, D. E., "A Lattice Model of Secure Information Flow," *Communications of the ACM*, Vol. 19, No. 5, 1976, pp. 236–243.

[6] Denning, D. E. and Denning, P. J., *Cryptography and Data Security*, Addison-Wesley, 1982.

[7] Object Management Group Inc., " The Common Object Request Broker : Architecture and Specification," Rev. 2.1, 1997.

[8] Merkl, D., Tjoa, A. M., and Vieweg, S., "BRANT - An Approach for Knowledge Based Document Classification Retrieval Domain," *Proc. of DEXA '92*, 1992, pp. 254–259.

[9] Sandhu, R. S., "Lattice-Based Access Control Models," *IEEE Computer*, Vol. 26, No. 11, 1993, pp. 9–19.

[10] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E., "Role-Based Access Control Models," *IEEE Computer*, Vol. 29, No. 2, 1996, pp. 38–47.

[11] Schneier, B., *Applied Cryptography*, John Wiley & Sons, 1996.

[12] Tachikawa, T., Yasuda, M., Higaki, H., and Takizawa, M., "Purpose-Oriented Access Control Model in Object-Based Systems," *Proc. of the 2nd Australasian Conf. on Information Security and Privacy (ACISP'97)*, 1997, pp. 38–49.

[13] Tari, Z. and Chan, S. W., "A Role-Based Access Control for Intranet Security," *IEEE Internet Computing*, Vol. 1, No. 5, 1997, pp. 24–34.