On the design of access control to prevent sensitive information leakage in distributed object systems: a Colored Petri Net based model

Panagiotis Katsaros[1]

[1] Department of Informatics, Aristotle University of Thessaloniki,
54124 Thessaloniki, Greece
katsaros@csd.auth.gr
http://delab.csd.auth.gr/~katsaros/index.html

Abstract. We introduce a Colored Petri Net model for simulating and verifying information flow in distributed object systems. Access control is specified as prescribed by the OMG CORBA security specification. An insecure flow arises when information is transferred from one object to another in violation of the applied security policy. We provide precise definitions, which determine how discretionary access control is related to the secure or insecure transfer of information between objects. The model can be queried regarding the detected information flow paths and their dependencies. This is a valuable mean for the design of multilevel mandatory access control that addresses the problem of enforcing object classification constraints to prevent undesirable leakage and inference of sensitive information.

1 Introduction

For a secure application it is not enough to control access to objects, without taking into account the information flow paths implied by a given, outstanding collection of access rights. The problem is faced by the use of *multilevel mandatory policies*, where users have no control and therefore they cannot be bypassed. Access to objects is granted on the basis of *classifications* (taken from a partially ordered set) assigned to objects and subjects requesting access to them.

The rigidity of these policies implies the need for a systematic design approach. In the design of mandatory access control we aim to enforce *object classification constraints* to prevent undesirable leakage and inference of sensitive information, while at the same time guaranteeing that objects will not be overclassified (to maximize information visibility). The set of constraints is decided based on the restrictions of the system's enterprise environment and on a view of the potential information flow paths. Our work aims to provide information regarding the system's structure, to be used in the design of appropriate mandatory access control.

In distributed object systems, objects interact by synchronous, asynchronous or deferred synchronous messages and detection of information flow is complicated by the presence of bi-directional information transfer between the senders and the receivers.

Information transfer from the sender to the receiver is accomplished through message parameters, while the opposite, through message reply. An information flow does not require direct message exchange between objects (*indirect information flow paths*). An object acquires information only when writing to its attributes and this operation results in one or more direct or indirect information flows.

We assume that access control is specified as prescribed by the OMG CORBA security specification. An *insecure flow* arises when information is transferred from one object to another in violation of the applied security policy.

We introduce a Colored Petri Net model to detect information flow paths based on system's object method dependencies and the applied access control. The model was implemented in CPN Tools ([7]), an advanced ML-based tool for editing, simulating and analyzing Colored Petri Nets.

To detect the insecure flow paths we implement the definitions we provide to determine how the applied discretionary access control is related to the secure or insecure transfer of information between objects. If there is an insecure flow to an object, we are often interested in enforcing an appropriate object classification constraint over all (and not only the insecure) sources of information flow to that object. In that case, we use the CPN Tools state space analysis functions to make queries regarding the detected flow paths.

Thus, the proposed model is a static analysis tool that provides data for the systematic design of multilevel mandatory access control. We preferred to use Colored Petri Net model checking and not to use Finite State Machine model checking because: (i) Colored Petri Nets possess the expressiveness and the formal analysis capacity of a Petri Net modeling language, (ii) they provide an explicit representation of both states and actions and at the same time retain the modeling flexibility provided in a programming language environment and (iii) Colored Petri Nets is a widespread modeling formalism with an easily accessible tool support that allows interactive simulation in an advanced graphical environment.

Section 2 provides a detailed description of the problem under consideration. Section 3 introduces the basic components of the information flow security model and provides the implemented definitions that determine when an information flow path is secure and when it is not secure. Section 4 focuses on the use of the CPN Tools state space analysis functions to make queries regarding the detected flows. Section 5 summarizes the latest related work reported in the bibliography and the paper concludes with a discussion on the potential impact of our work.

2 Basic definitions and problem statement

Distributed object systems are composed of a set of objects $o_1, o_2, \ldots, o_n$, which interact to accomplish common goals. An object's methods $op_l^{o_i}$, $1 \leq l \leq$ #(methods of o_i), $1 \leq i \leq n$, are invoked by synchronous, asynchronous or deferred synchronous method requests.

A *message msg* is defined as a pair (*method*, *type*), with

$$type \in \{s \mid \text{synchronous invocation request}\} \cup \{drq \mid \text{deferred synchronous invocation request}\} \cup \{a \mid \text{asynchronous invocation request}\} \cup \{drp \mid \text{deferred synchronous invocation reply}\}$$

and

$$method \in \bigcup_i \left\{ op_l^{o_i} \mid 1 \le l \le \#(\text{methods of } o_i) \right\} \cup \{read, write\}$$

where *read*, *write* correspond to primitive synchronous messages ($type = s$) that are sent by an object to itself to read or respectively update its state.

A *message sequence specification* $MsgSeq(op_l^{o_i})$ for method $op_l^{o_i}$, $1 \le l \le$ #(methods of o_i) is a total order relation $\Rightarrow$ over the set

$$\{msg_s \mid 1 \le s \le \#(\text{messages in } MsgSeq(op_l^{o_i}))\}$$

of nested method invocations, as well as read and/or write messages generated by $op_l^{o_i}$. For every two msg_s, $msg_t \in MsgSeq(op_l^{o_i})$, $msg_s \Rightarrow msg_t$ if and only if msg_s *is sent/received before* msg_t. We note that $\Rightarrow$ defines an *invocation order* that does not necessarily coincide with the order in which method executions are completed.

To make it clear, if in $MsgSeq(op_l^{o_i})$ holds that $(op_k^{o_j}, drq) \Rightarrow (write, s)$, this does not mean that $op_k^{o_j}$ *is executed before write*. However, if

$$(op_k^{o_j}, drq) \Rightarrow (op_k^{o_j}, drp) \Rightarrow (write, s)$$

i.e. if the reply of a deferred synchronous request to o_j is received before *write*, then $op_k^{o_j}$ is executed before *write*. In that case, if $read \in MsgSeq(op_k^{o_j})$ and the used credentials include the read access right for o_j, then there is an information flow from o_j to o_i. Moreover, o_j may also be the source of additional indirect flows to other objects, if for example $op_l^{o_i}$ has been synchronously invoked by another object method.

An information flow is *insecure*, if the derived information is transferred to the target, in violation of the applied security policy. An information flow takes place *even if the information written to the target is not the same as the information read*, but is derived from it by executing computations.

We are primarily interested in detecting insecure flow paths and enforcing appropriate object classification constraints to prevent undesirable leakage of information. However, this is not enough. It is also necessary to ensure compliance with potential business constraints regarding undesirable inference of sensitive information. In a service-outsourcing environment (e.g. web services) these are key concerns that have to be satisfied. Thus, we also need to query the model about the detected "secure" information flow paths to an object.

The design problem under consideration is addressed by the implementation of an appropriate multilevel mandatory policy. This policy is based on the assignment of *access classes* to objects and subjects and is used in conjunction with the applied discretionary access control. Access classes in a set L are related by a partial order, called *dominance relation* and denoted by $\ge$. The dominance relation governs the visibility of information: a subject has read access only to the objects classified at the subject's level or below and is possible to perform write operations only to the objects classified above the subject's level. The expression $x \ge y$ is read as "*x dominates y*".

The partially ordered set $(L, \geq)$ is assumed to be a *lattice*. We refer to the maximum and minimum elements of a lattice as $\top$ (top) and $\bot$ (bottom). Figure 1 depicts an example classification lattice.

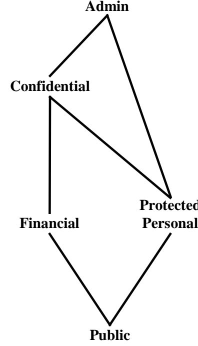

Fig. 1. An example security lattice

In general, the security level to be assigned to an object depends on the sensitivity of the data in its state. If an object's state is related to publicly accessible customer data this object might be labeled at a level such as `Public`. If an object's state is related to customer income data, this object might be labeled at a higher level, such as `Financial`.

However, the design of multilevel mandatory policies can be done in a systematic manner, based on a set of classification constraints. These constraints specify the requirements that the security levels assigned to objects must satisfy. They are constraints on a mapping λ: $\{o_i \mid 1 \leq i \leq n\} \rightarrow L$ that assigns to each object a security level $l \in L$, where $(L, \geq)$ is a classification lattice. Object classification constraints, like for example when $\lambda(o_i) \geq$ `Financial`, are used to ensure that objects are assigned security levels high enough to protect their state.

If our model detects an "insecure" information flow from o_j to o_m, the flow will take place only when the subjects *clearance level* dominates the classification of o_j and is dominated by the classification of o_m. In that case, the constraint $\lambda(o_m) \geq \lambda(o_j)$ prevents sensitive information leakage in a low-classified object (the opposite prevents the occurrence of the detected information flow in all cases). If there is also an additional flow from o_s to o_m, both of them take place only when the subjects clearance level dominates the least upper bound (`lub`) of the classifications of o_s and o_j. In that case, the constraint

$$\texttt{lub}\{\lambda(o_s), \lambda(o_j)\} \geq \texttt{Financial}$$

will ensure that both flows will take place only when the subjects clearance level dominates a given ground level. In this way it is possible to prevent potential inference of high-classified data from low-classified objects.

The proposed model is queried regarding the "insecure" and the "secure" information flow paths, in order to direct the specification of constraints, wherever there is a need *to prevent undesirable information leakage or information inference*. This al-

lows to avoid overclassification and thus to maximize information visibility. It is then possible to implement an appropriate mandatory policy after having solved the derived set of constraints. A recently published algorithm is the one described in [3], but it is not the only one published.

3 The Colored Petri Net based model

Colored Petri Nets (CP-nets) provide us the primitives for the definition of diverse data types (such as privilege attributes, method names, object ids and others) and the manipulation of their data values, while retaining the expressiveness and the formal analysis capacity of a Petri Net modeling language.

The formal semantics of CP-nets is outlined in Appendix. Model states are represented by means of *places* (which are drawn as ellipses). Each place has an associated *data type* determining the kind of data, which the place may contain (by convention the type information is written in italics, next to the place). The type declarations implicitly specify the operations that can be performed on the values of the types. A state of a CP-net is called a *marking* and consists of a number of *tokens* positioned on the individual places. Each token carries a data value, which belongs to the type of the corresponding place.

A marking of a CP-net is a function, which maps each place into a *multi-set of tokens* (see the Appendix) of the correct type. We refer to the token values as *token colors* and to their data types as *color sets*. The types can be arbitrarily complex, e.g., a record where one field is a real, another field is a text string and a third field is a list of integers.

CP-net actions are represented by means of *transitions*, which are drawn as rectangles. An incoming arc indicates that the transition may remove tokens from the corresponding place while an outgoing arc indicates that the transition may add tokens. The exact number of tokens and their data values are determined by *arc expressions*, which are positioned next to the arcs. Arc expressions may contain variables as well as constants. To talk about the *occurrence of a transition*, we need to bind incoming expressions to values from their corresponding types. Let us assume that we bind the incoming variable v of some transition T to the value d. The pair $(T, <v = d>)$ is called *binding element* and this binding element is *enabled* in a marking M, when there are enough tokens in its input places. In a marking M, it is possible to have enabled more than one binding elements of T. If the binding element $(T, <v = d>)$ occurs, it removes tokens from its input places and adds tokens to its output places. In addition to the arc expressions, it is possible to attach a boolean expression with variables to each transition. This expression is called *guard* and specifies that we only accept binding elements for which the expression evaluates to true.

The behavior of a CP-net is characterized by a set of dynamic properties:

- *Bounds-related properties* characterize the model in terms of the number of tokens we may have at the places of interest.
- *Home properties* provide information about markings or sets of markings to which it is always possible to return.

- *Liveness properties* examine whether a set of binding elements *X* remains active: "For each reachable marking *M´*, is it possible to find a finite sequence of markings starting in *M´* that contain an element of *X*?"
- *Fairness properties* provide information about how often the different binding elements occur.

CP-nets are analyzed, either by

- simulation,
- formal analysis methods such as the construction of *occurrence graphs*, which represent all reachable markings,
- calculation and interpretation of system *invariants* (called place and transition invariants),
- performance of *reductions* which shrink the net without changing a certain selected set of properties and
- the check of structural properties, which guarantee certain behavioral properties.

In CPN Tools, CP-nets are developed in a modern GUI-based environment that provides interactive feedback for the model's behavior through simulation. Colors, variables, function declarations and net inscriptions are written in CPN ML, which is an extension of Standard ML and for this reason employs a functional programming style. In CPN Tools we employ simple as well as compound color sets such as product, record, list and union color sets.

The toolset provides the necessary functionality for the analysis of simple and *timed CP-nets* specified in a number of *hierarchically related pages*. Typical models consist of 10-100 pages with varying complexity and programming requirements. The companion *state space tool* allows the generation of the entire or a portion of the model's state space (occurrence graph) and the performance of standard as well as non-standard analysis queries.

In our model, *the CP-net structure depends on the system's object method dependencies*. These dependencies may be derived from the system's source code with a code-slicing tool ([11]). Taking into account that in CPN Tools the net is stored in an XML-based format, we believe that models can be automatically generated using an appropriate XML text generator.

3.1 The CORBA Security model

Distributed object systems typically support a large number of objects. CORBA Security ([13]) provides abstractions to reduce the size of access control information and at the same time to allow fine-grained access to individual operations rather than to the object as a whole. Access policies are defined based on privilege and control attributes and access decisions are made via the standard access decision interface that is modeled by the CP-net we present here.

Principals are users or processes accountable for the actions associated with some user. In a given security policy, each principal possesses certain *privilege attributes* that are used in access control: such attributes may be access identities, roles, groups, security clearance and so on. At any time, a principal may choose to use only a subset

of the privilege attributes it is permitted to use, in order to establish its rights to access objects.

Access control is defined at the level of individual object invocations. The access decision function bases its result on the current privilege attributes of the principal, the operation to be performed and the *access control attributes* of the target object.

A set of objects where we apply common security policies is called *security policy domain*. Security domains provide leverage for dealing with the problem of scale in policy management. The CORBA Security specification allows objects to be members of multiple domains: the policies that apply to an object are those of all its enclosing domains. CORBA Security does not prescribe specific *policy composition rules*. Such rules are the subject of the system's security design and this allows for potentially unlimited flexibility in combining complementary access control policies.

A *domain access policy* grants a set of subjects the specified set of rights to perform operations on all objects in the domain. In Table 1 we provide a sample domain access policy. As subject entries we use the privilege attributes possessed by the principals. Thus, user identities can be considered to be a special case of privilege attributes. In CORBA Security, rights are qualified into sets of "access control types", known as *rights families*. There is only one predefined rights family that is called `corba` and contains the three rights `g` (for get or read), `s` (for set or write) and `m` (for manage).

Table 1. Domain access policy (granted rights)

Privilege Attribute	Domain	Granted Rights
access_id: a1	1	corba: gs-
access_id: a2	2	corba: g--
group: g1	1	corba: g--
group: g1	2	corba: gs-
group: g2	1	corba: gs-

Rights to privilege attributes are granted by an `AccessPolicy` object. An operation of a secure object can be invoked only when the principal possesses the set of rights prescribed by the `RequiredRights` object. Table 2 shows an example `RequiredRights` object that defines the rights required to gain access to each specific method of an object. There is also a mechanism to specify whether a user needs all the rights - in a method's required rights entry - to execute that method (AND semantics) or whether it is sufficient to match any right within the entry (OR semantics).

Table 2. Required rights

Required Rights	Rights Combinator	Operation	Interface (class)
corba: g--	all	M1	c_1
corba: g--	all	M3	
corba: gs-	all	M4	c_2
corba: -s-	all	M0	
corba: -s-	all	M2	
corba: gs-	any	M5	c_3

Table 3. Domain memberships and object classes

Object	Domain
o_1 , o_2, o_5, o_{12}	d_1
o_8, o_9	d_2

Objects	Class
o_1 , o_8	c_1
o_2 , o_5, o_9	c_2
o_{12}	c_3

Table 3 specifies the security domain memberships and the object classes, for the case access control introduced in Tables 1 and 2.

The `AccessDecision` object determines the validity of invocation requests based on the privilege and control attributes provided by the `AccessPolicy` and `RequiredRights` objects. There are no explicit rules on how to calculate the access decision: CORBA Security does not prescribe how an `AccessPolicy` object combines rights granted by different privilege attribute entries (when a subject has more than one privilege attribute to which the `AccessPolicy` grants rights). Taking into account the absence of policy composition rules for domain hierarchies, all these make feasible the implementation of different access decision functions and thus allow for potentially unlimited flexibility in security policy specification.

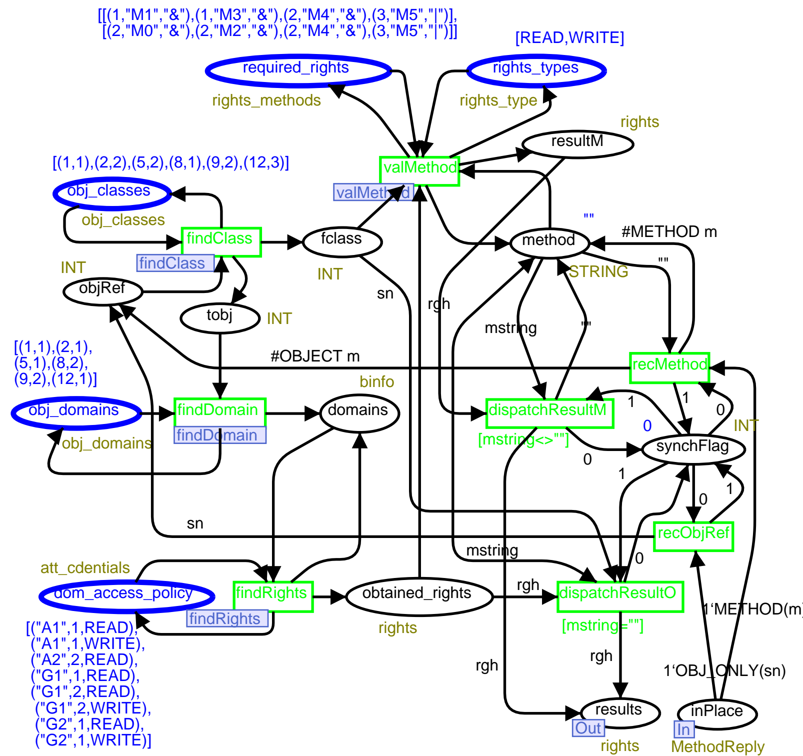

Fig. 2. The access decision CP-net submodel

Figure 2 presents the top-layer of the CP-net submodel implementing the following access decision function: "A method *m* can be executed if the requester's rights match the rights specified in the method's entry in the `RequiredRights` table". The shown CP-net is used in the following ways:

- To obtain privilege attributes and access rights (output place `results`) to proceed to the execution of the method specified in the union typed place `inPlace` (if any).

- To derive the *access control list* (list of privilege attribute and access right pairs in output place `results`) for the object specified in place `inPlace` (if any).

The domain access policy (bold place `dom_access_policy`) is specified as a list of triads, which respectively represent privilege attribute, domain number and right. The required rights table is given as lists of triads (bold place `required_rights`), which respectively represent class number, method name and rights combinator and each ML list refers to the corresponding right of the ML list shown in the `rights_types` bold place. The data shown in Table 3 determine the initial markings of the bold places `obj_domains` and `obj_classes`.

Due to space limitations we omit the description of the low-level CP-nets implementing the hierarchically related substitution transitions `valMethod`, `findClass`, `findDomain` and `findRights`.

3.2 The information flow security model

The CP-net of Figure 2 corresponds to the `protSys` substitution transition of Figure 3 that mimics a method execution: an object sends the messages of method's sequence specification (given in the bold input/output place `obj`) to itself or to other objects. Access to the object's state is accomplished by dispatching primitive read and write messages to itself: each of them is supposed to be executed synchronously.

In synchronous and deferred synchronous communication (hierarchically related substitution transitions `doSynchSend` and `doDSynchSend`) a reply is eventually returned, together with a list of all object identifiers (color `binfo` for the place `AOsL`) where read operations were allowed by the used access control. This list is termed as *Accessed Objects List* (AOsL). AOsL list is transmitted forward (requests) and backward (replies) as prescribed by methods' message sequence specifications, in order to record the performed read operations in all accessed objects.

An information flow to an object takes place only when information is written to it (substitution transition `doWrite`). In that case, there is an information flow from each one of the objects contained in the transmitted AOsL list. However, *not all of them violate the applied access control*:

Definition 3.1

An information flow from an object o_i (source) to an object o_j (target) *is not secure* (may cause undesirable information leakage), if the privilege attributes that grant read access to the target are not a subset of the set of attributes, which grant read access to the source.

Definition 3.2

An information flow to an object o_j *is secure*, if the privilege attributes that grant read access to it are also contained in all sets of privilege attributes, which grant read access to the objects contained in the transmitted AOsL list.

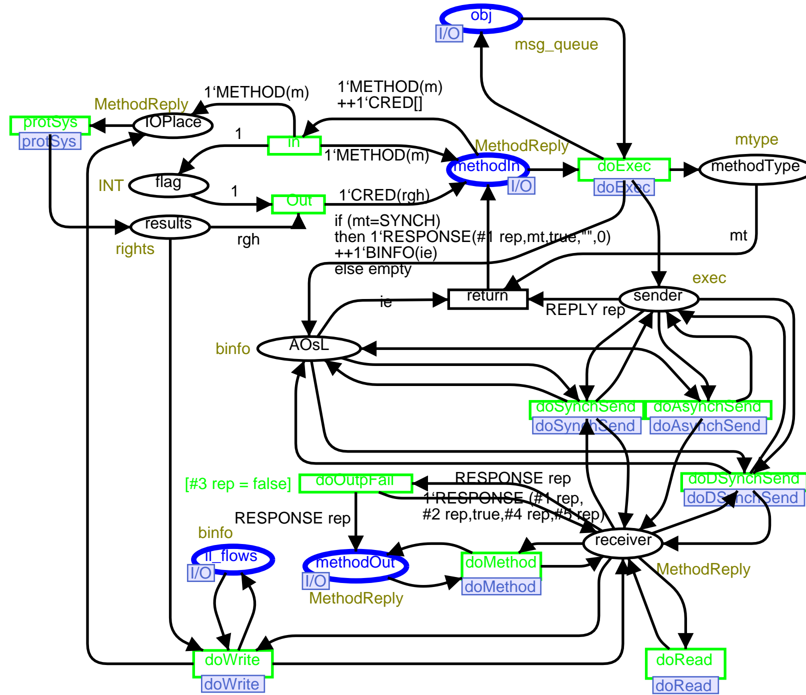

Fig. 3. The top layer of the method execution CP-net

```
color mtype                 = with SYNCH | ASYNCH | DSYNCH | DREP;
color crtype                = with READ | WRITE;
color rights_type           = list crtype;
color INT                   = int;
color BOOL                  = bool;
color STRING                = string;
color MSG_SNxSTATUS         = product INT * STRING;
color msg_rec               = record NUMBER:INT * METHOD:STRING
                               * TYPE:mtype * OBJECT:INT;
color right                 = product STRING*crtype;
color rights                = list right;
color binfo                 = list INT;
color msg_queue             = list msg_rec;
color reply                 = product INT * mtype * BOOL * STRING * INT;
color strLst                = list STRING;
color MethodReply           = union   RESPONSE:reply+
                              METHOD:msg_rec+
                              CRED:rights+
                              ACL:strLst+
                              BINFO:binfo+
                              OBJ_ONLY:INT;
color exec                  = union REPLY:reply +
                              MSG_QUEUE:msg_queue +
                              CREDENT:rights;

var q,p, messages           : msg_queue;
var m                       : msg_rec;
var sn,sm,k                 : INT;
var mt                      : mtype;
var rep                     : reply;
var ie,ic                   : binfo;
var rgh                     : rights;

fun aux2 k l                = if cf(k,l)>0 then nil else [k];
fun unio (x::xl,yl)         = (aux2 x yl)++unio(xl,yl)
                            | unio (_,yl) = yl;
```

Fig. 4. Colors, variables and functions used in CP-net incriptions

Figure 4 summarizes the color, variable and function declarations used for the transition and arc inscriptions of the CP-nets of Figures 3 and 5.

Figure 5 reveals the details of the `doSynchSend` substitution transition of Figure 3. Method execution is blocked (place `RmvMessage`) up to the reception of the expected reply - signaled by the token `(sn,"REPLIED")` in place `NextSend`. AOsL list is returned (arc inscription `1'BINFO(ic)`) together with the method reply. Then, AOsL list is updated as appropriate (function inscription `unio(ie,ic)` to calculate the union of the existing list `ie` and the received list `ic`).

We omit the details of the `doAsynchSend` and `doDSynchSend` transitions shown in Figure 3, but we stress the fact that AOsL list is never changed as a result of an asynchronous method execution.

Fig. 5. The CP-net for the `doSynchSend` substitution transition shown in Figure 3

A system model is composed of a number of interacting instances of the CP-net shown in Figure 3 and each instance represents a particular method execution. For all method executions their instance input places (`obj`, `methodIn`) are initialized as imposed by the system's object method dependencies. Insecure information flow paths are detected at the `doWrite` substitution transition and for each object are separately recorded at the `il_flows` output places.

Figure 6 reports the simulation results given for the access control of Figure 2 and the case system model:

$$MsgSeq(M0^{o_2}) = \{(M1^{o_1}, s) \Rightarrow (M2^{o_9}, s) \Rightarrow (M3^{o_8}, s) \Rightarrow (write, s)\}$$

$$MsgSeq(M1^{o_1}) = \{(read, s)\}$$

$$MsgSeq(M2^{o_9}) = \{(M4^{o_5}, s) \Rightarrow (write, s)\}$$

$$MsgSeq(M3^{o_8}) = \{(read, s)\}$$

$$MsgSeq(M4^{o_5}) = \{(read, s) \Rightarrow (write, s)\}$$

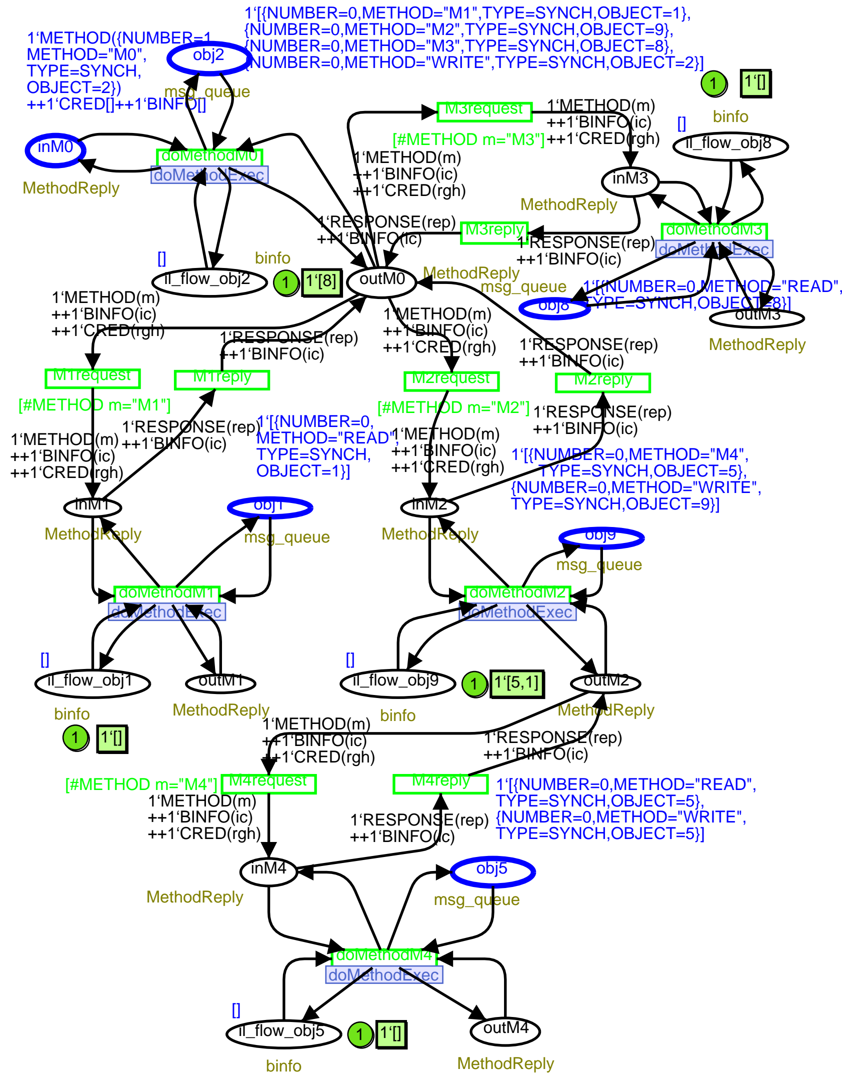

Fig. 6. A case system model and the detected insecure information flow paths

Insecure information flow paths are detected and recorded in `il_flow_obj2` and `il_flow_obj9` places. We observe the existence of flows from o_1 and o_5 to o_9 and from o_8 to o_2. We note that method execution control flow, like for example conditional execution, is not taken into account. We are interested for the detection of all potential information flow paths and we want to take them into account, in order to ensure compliance with business constraints regarding undesirable leakage or inference of sensitive information.

The simulated model reports the verification results regarding the detection of insecure flow paths. However, this is not enough if we want to use it as a mean to guide the specification of object classification constraints.

4 State space analysis

Analysis of the occurred information flow paths is performed after having generated all possible states that the system can reach. This can be done, by exploiting the CPN Tools state space analysis facilities ([7]).

Given the full state space, also known as *occurrence* or *reachability graph*, we can check the standard properties mentioned in section 3, as well as the existence of an *occurrence sequence* (reachability) to a particular marking (state). Figure 7 summarizes the results for the standard checks mentioned in section 3. The full state space is generated in 13 secs and consists of 2029 nodes (states) and 3282 arcs. There are no live transition instances and infinite occurrence sequences and the single dead marking that reflects the completed execution of method "M0" corresponds to the node number 2029.

```
Statistics
---------------------------------------------
  Occurrence Graph
    Nodes:  2029
    Arcs:   3282
    Secs:   13
    Status: Full

 Boundedness Properties
---------------------------------------------
  Best Integers Bounds    Upper      Lower
  NewPage'il_flow_obj1 1  1          1
  NewPage'il_flow_obj2 1  1          1
  NewPage'il_flow_obj5 1  1          1
  NewPage'il_flow_obj8 1  1          1
  NewPage'il_flow_obj9 1  1          1
  . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . .

 Home Properties
---------------------------------------------
  Home Markings:  [2029]

 Liveness Properties
---------------------------------------------
  Dead Markings:  [2029]
  Live Transitions Instances: None

 Fairness Properties
---------------------------------------------
No infinite occurrence sequences.
```

Fig. 7. The state space analysis standard report for the CP-net of Figure 6

Model querying regarding all detected information flow paths is performed through evaluation of simple ML functions. The set of predefined ML functions used to explore the generated state space is summarized in Table 4.

Table 5 shows (in the result column) the information flow data derived for the case system model.

Table 4. State space querying functions

function description	use
Mark.<PageName>'<PlaceName> N M	Returns the set of tokens positioned on place <PlaceName> on the Nth instance of page <PageName> in the marking M
ListDeadMarkings ()	Returns a list with all those nodes that have no enabled binding elements.
hd l	Returns the head of l.
SearchNodes (<search area>, <predicate function>, <search limit>, <evaluation function>, <start value>, <combination function>)	Traverses the nodes of the part of the occurrence graph specified in <search area>. At each node the calculation specified by <evaluation function> is performed and the results of these calculations are combined as specified by <combination function> to form the final result. The <predicate function> maps each node into a boolean value and selects only those nodes, which evaluate to true. We use the value EntireGraph for <search area> to denote the set of all nodes in the occurrence graph and the value 1 for <start value> to continue the search until the first node, for which the predicate function evaluates to true.

Table 5. Information flow security queries

1. Insecure information flow sources:		
object id	function	result
o_2	Mark.NewPage'il_flow_obj2 1 (hd (ListDeadMarkings( )))	val it = [[8]]: binfo ms
o_1	Mark.NewPage'il_flow_obj1 1 (hd (ListDeadMarkings( )))	val it = [[]]: binfo ms
o_8	Mark.NewPage'il_flow_obj8 1 (hd (ListDeadMarkings( )))	val it = [[]]: binfo ms
o_9	Mark.NewPage'il_flow_obj9 1 (hd (ListDeadMarkings( )))	val it = [[5,1]]: binfo ms
o_5	Mark.NewPage'il_flow_obj5 1 (hd (ListDeadMarkings( )))	val it = [[]]: binfo ms

2. All information flow paths to o_2 (including the "secure" ones):	
function	result
Mark.doWrite'recBINFO 1 (hd (SearchNodes (EntireGraph, fn n => (Mark.doWrite'recBINFO 1 n <> empty andalso (Mark.doWrite'rstrrights 1 n <> empty andalso Mark.doWrite'torrights 1 n <> empty)), 1, fn n => n, [], op::)))	val it = [[1,5,8]]: binfo ms

3. Privilege attributes for read access to o_2:	
Mark.doWrite'torrights 1 (hd (SearchNodes (EntireGraph, fn n => (Mark.doWrite'recBINFO 1 n <> empty andalso (Mark.doWrite'rstrrights 1 n <> empty andalso Mark.doWrite'torrights 1 n <> empty)), 1, fn n => n, [], op::)))	val it = [["A1", "G1", "G2"]]: strLst ms

4. Privilege attributes for read access to insecure source o_8:	
Mark.doWrite'rstrrights 1 (hd (SearchNodes (EntireGraph, fn n => (Mark.doWrite'recBINFO 1 n <> empty andalso (Mark.doWrite'rstrrights 1 n <> empty andalso Mark.doWrite'torrights 1 n <> empty)), 3, fn n => n, [], op::)))	val it = [["A2", "G1"]]: strLst ms

The results of query 1 verify the simulation results of Figure 6 regarding the insecure flow paths detected at the found dead marking. The query is based on inspection of the marking of `il_flow` places on the first instance of page `NewPage`, for the head of the list of dead markings, which in fact contains the single dead marking with node number 2029.

In query 2 we use the function `SearchNodes` to explore the entire state space for a marking that yields all flows (including the "secure" ones) to o_2. Queries 3 and 4 reveal the details of the insecure flow (definition 3.1) sourced at o_8.

Function `SearchNodes` is used as a tool of potentially unlimited flexibility in querying the model regarding the "insecure" and the "secure" information flow paths, in order to direct the specification of object classification constraints. Alternatively, CPN Tools includes a library for defining queries in a CTL-like temporal logic.

State spaces grow exponentially, with respect to the number of independent processes. In the proposed model, this problem becomes evident, when using asynchronous and/or deferred synchronous method calls. From the alternative published analyses our model fits to the modular state space analysis described in [2]. The behavior of the entire system can be captured by the state spaces of the modules corresponding to individual method executions (Figure 6), combined with an appropriate synchronization graph. Unfortunately, CPN Tools does not currently support the generation of separate state space modules and the required synchronization graph. Thus, the application of the forenamed analysis approach remains an open research prospect.

5 Related work

Information flow security is an active research problem that was first approached in 1973 ([10]) and that is still attracting the interest of researchers in a number of recently published works ([12], [1], [6], [4], [5]).

Recent research works in the context of distributed object systems ([6], [16] and [14])

- are based on different and often not realistic assumptions on when an information flow occurs,
- do not always take into account that in real systems, methods are invoked in a nested manner,
- are bound to specific role-based or purpose-oriented access control models and none employs the CORBA Security reference model or
- aim in the dynamic control of information flow by the use of an appropriate run-time support that in most systems is not available.

Our work (i) takes into account the bi-directional nature in the direct or indirect information transfer between the senders and the receivers, (ii) allows for modeling nested object invocations, (iii) employs the CORBA Security reference model and for this reason is not bound to a specific access control model and (iv) aims in the static verification of information flow security and for this reason does not assume proprietary run-time support. Moreover, it is based on a widespread modeling formalism with an easily accessible advanced tool support.

Other interesting sources of related work are the introduction into lattice-based access control that was published in [15] and the mandatory access control that is specified in [9], by the use of the CORBA Security reference model.

6 Conclusion

In modern networked business information systems and in service-based business activities, where different customers are concurrently using the provided services, compliance with business constraints regarding undesirable inference of sensitive information is a central design issue.

The problem under consideration is addressed by the implementation of an appropriate multilevel mandatory policy. However, the rigidity of these policies implies the need for a systematic design approach. We introduced a Colored Petri Net model that simulates and detects "insecure" information flow paths according to the given definitions determining when a flow path is not secure The model can be queried regarding the existing ("insecure" and "secure") flows, in order to direct the specification of object classification constraints, wherever there is a need of them. This allows to avoid overclassification and thus to maximize information visibility.

Acknowledgments

We acknowledge the CPN Tools team at Aarhus University, Denmark for kindly providing us the license of use of the valuable CP-net toolset.

References

1. Chou, S.-C.: Information flow control among objects: Taking foreign objects into control, In: Proceedings of the 36th Hawaii International Conference on Systems Sciences (HICSS'03), IEEE Computer Society (2003) 335a-344a
2. Christensen, S., Petrucci, L.: Modular state space analysis of Coloured Petri Nets, In: Proceedings of the 16th International Conference on Application and Theory of Petri Nets, Turin, Italy (1995) 201-217
3. Dawson, S., Vimercati, S., Lincoln, P., Samarati, P.: Maximizing sharing of protected information, Journal of Computer and System Sciences 64 (3), (2002) 496-541
4. Georgiadis, C., Mavridis, I., Pangalos, G.: Healthcare teams over the Internet: Programming a certificate-based approach, International Journal of Medical Informatics 70 (2003) 161-171
5. Halkidis, S. T., Chatzigeorgiou, A., Stephanides, G.: A qualitative evaluation of security patterns, In: Proceedings of ICICS 2004, LNCS 3269, Springer-Verlag (2004) 132-144
6. Izaki, K., Tanaka, K., Takizawa, M.: Information flow control in role-based model for distributed objects, In: Proceedings of the 8th International Conference on Parallel and Distributed Systems (ICPADS'01), Kyongju City, Korea, IEEE Computer Society (2001) 363-370

7. Jensen, K.: An introduction to the practical use of colored Petri Nets, In: Lectures on Petri Nets II: Applications, LNCS, Vol. 1492 (1998) 237-292
8. Jensen, K.: An introduction to the theoretical aspects of colored Petri Nets, In: A Decade of Concurrency, LNCS, Vol. 803 (1994) 230-272
9. Karjoth, G.: Authorization in CORBA security, In: ESORICS'98, LNCS, Vol. 1485 (1998) 143-158
10. Lampson, B. W.: A note on the confinement problem, Communication of the ACM 16 (10), (1973) 613-615
11. Larsen, L., Harrold, M.: Slicing object oriented software, In: Proceedings of the 18th International Conference on Software Engineering (1996) 495–505
12. Masri, W., Podgurski, A., Leon, D.: Detecting and debugging insecure information flows, In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, Bretagne, France, IEEE Computer Society (2004) 198-209
13. Object Management Group: Security service specification, version 1.7, OMG Document 99-12-02 (1999)
14. Samarati, P., Bertino, E., Ciampichetti, A., Jajodia, S.: Information flow control in object-oriented systems, IEEE Transactions on Knowledge and Data Engineering 9 (4), (1997) 524-538
15. Sandhu, R. S.: Lattice-based access control models, IEEE Computer 26 (11), (1993) 9-19
16. Yasuda, M., Tachikawa, T., Takizawa, M.: Information flow in a purpose-oriented access control model, In: Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS'97), Seoul, Korea, IEEE Computer Society (1997) 244-249

Appendix

In this section, we outline the formal semantics of CP-nets, as they are defined in [8].

Definition 1 A multi-set m, over a non-empty set S is a function $S \rightarrow \aleph$ represented as a sum

$$\sum_{s \in S} m(s)`s$$

By S_{MS} we denote the set of all multi-sets over S. The non-negative integers $\{m(s) | s \in S\}$ are the coefficients of the multi-set.

Definition 2 A Colored Petri Net (CP-net) is a tuple CPN=(Σ, P, T, A, N, C, G, E, I) where:

(i) Σ is a finite set of non-empty types, also called color sets
(ii) P is a finite set of places (drawn as ellipses)
(iii) T is a finite set of transitions (drawn as rectangles)
(iv) A is a finite set of arcs
(v) N is a node function $A \rightarrow P \times T \cup T \times P$
(vi) C is a color function $P \rightarrow \Sigma$
(vii) G is a guard function that maps each transition $t \in T$ into a Boolean expression where all variables have types that belong to Σ:

$$\forall t \in T: \text{Type}(G(t)) = B \wedge \text{Type}(\text{Var}(G(t))) \subseteq \Sigma$$

(viii) E is an arc expression function that maps each arc $a \in A$ into an expression that is evaluated in multi-sets over the type of the adjacent place p:

$$\forall a \in A: \text{Type}(E(a))=C(p)_{MS} \wedge \text{Type}(\text{Var}(E(a))) \subseteq \Sigma, \text{ with } p=N(a)$$

(ix) I is an initialization function that maps each place $p \in P$ into a closed expression of type $C(p)_{MS}$:

$$\forall p \in P: \text{Type}(I(p))=C(p)_{MS}$$

When we draw a CP-net we omit initialization expressions, which evaluate to $\emptyset$.

The set of arcs of transition t is

$$A(t) = \{a \in A \mid N(a) \in P \times \{t\} \cup \{t\} \times P\}$$

and the variables of transition t is

$$\text{Var}(t) = \{v \mid v \in \text{Var}(G(t)) \vee \exists a \in A(t): v \in \text{Var}(E(a))\}$$

Definition 3 A binding of a transition t is a function b defined on Var(t), such that:

(i) $\forall v \in \text{Var}(t): b(v) \in \text{Type}(v)$

(ii) The guard expression G(t) is satisfied in binding b, i.e. the evaluation of the expression G(t) in binding b - denoted as $G(t)<b>$ - results in true.

By B(t) we denote the set of all bindings for t.

Definition 4 A token element is a pair (p, c) where $p \in P$ and $c \in C(p)$. A binding element is a pair (t, b) where $t \in T$ and $b \in B(t)$. The set of all token elements is denoted by TE and the set of all binding elements is denoted by BE.

A marking is a multi-set over TE and a step is a non-empty and finite multi-set over BE. The initial marking M_0 is the marking, which is obtained by evaluating the initialization expressions:

$$\forall (p,c) \in TE: M_0(p,c)=(I(p))(c)$$

The set of all markings and the set of all steps are denoted respectively by M and Y.

For all $t \in T$ and for all pairs of nodes $(x_1, x_2) \in (P \times T \cup T \times P)$ we define

$$A(x_1, x_2) = \{a \in A \mid N(a) = (x_1, x_2)\} \text{ and } E(x_1,x_2) = \sum_{a \in A(x_1,x_2)} E(a)$$

Definition 5 A step Y is enabled in a marking M if and only if

$$\forall p \in P: \sum_{(t,b) \in Y} E(p,t)<b> \leq M(p)$$

We then say that (t,b) is enabled and we also say that t is enabled. The elements of Y are concurrently enabled (if $|Y| \geq 1$).

When a step Y is enabled in a marking M_1 it may occur, changing the marking M_1 to another marking M_2, defined by:

$$\forall p \in P: M_2(p) = (M_1(p) - \sum_{(t,b) \in Y} E(p,t)<b>) + \sum_{(t,b) \in Y} E(t,p)<b>$$

M_2 is directly reachable from M_1.