Simulation and verification of information flow paths for access control policies specified in the CORBA Security setting

Panagiotis Katsaros

Department of Informatics, Aristotle University
Thessaloniki, 54124, Greece
email: {katsaros}@csd.auth.gr

Abstract. The OMG CORBA security specification defines the core facilities and interfaces for ensuring the required level of security in CORBA-compliant systems. However, 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 requirement to prevent insecure information leakage among objects is a key concern that has to be satisfied. We describe a Colored Petri Net model that allows simulating and verifying information flow security for access control policies specified in the OMG CORBA Security setting. The proposed model possesses the virtue of simulating insecure information leakage in a graphical environment and allows querying about the detected information flow paths and their dependencies.

Index terms – CORBA security, information flow security, access control, Colored Petri Nets, verification

1 Introduction

Most access control mechanisms are designed to control access to objects without any constraint on deriving information from one object and in subsequent operations on transferring it to other objects (e.g. [1]). An insecure flow arises when information is transferred from one object to another, in violation of the applied security policy.

The requirement to prevent insecure information leakage among objects is a key concern that has to be satisfied. This is done by the design of a mandatory access control over which, users have no control and therefore it cannot be bypassed. The design of access control can be based on a system model used for the detection of insecure information leakage and for querying about the detected information flow paths and their dependencies.

We propose a Colored Petri Net model that allows simulating and verifying information flow security when access control is specified as in the OMG CORBA Security ([11]) setting. The most notable benefit of having preferred the Colored Petri Net formalism and not a state-machine based formalism is that Colored Petri Nets possess the expressiveness and the formal analysis capacity of a Petri Net based modeling language. 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: Colored Petri Nets offer the primitives for the definition of diverse data types and the manipulation of their data values. It is a widespread modeling formalism with an easily accessible advanced tool support that allowed us to simulate insecure information leakage in a graphical environment and to query the generated state space about the detected information flow paths and their dependencies.

The model is implemented in CPN Tools ([5]), an ML-based tool for editing, simulating and analyzing Colored Petri Nets. The model's structure depends on the object method dependencies, which can be easily derived from the system's source code by a code-slicing tool ([9]). Since in CPN Tools models are stored in an XML-based format, we believe that model building can be fully automated by the use of an appropriate XML text generator.

The CORBA Security model is characterized by sufficient generality for expressing typical discretionary access control policies, as well as lattice-based access control ([7]) and role-based access control ([1]) policies. It makes use of appropriate abstractions that result in reduced size access control data and at the same time allow fine-grained access to individual operations rather than to the object as a whole.

The information flow security model views the system as a set of objects, which communicate with the other objects via messages and their replies. We allow for synchronous as well as asynchronous and deferred synchronous communication. When an object receives a message, the corresponding method is executed. A method execution can require the object to send a message to itself (for a read or write operation) or to another object. A reply is eventually returned to the object, which sent the message, apart from the asynchronous communication case. Accesses to the attributes of an object are accomplished only via primitive read and write messages to it.

A read message causes an information flow path from the object sent, if the read operation is (allowed to be) executed. A write message results in one or more information flow paths to the object sent, if the passed information is (allowed to be) written into the object. An information flow does not require direct message exchange between objects (indirect information flow paths). Our model determines whether the detected information flow paths are complied with the applied access control (are secure) or not.

Section 2 is a brief presentation of the Colored Petri Net formalism and the CPN Tools toolset. Section 3 introduces the CORBA authorization model and describes the proposed information flow security model. Section 4 is focused on the provided state space based verification functionality. Section 5 summarizes the latest developments in the related bibliography and the paper concludes with a discussion on the potential impact of our work.

2 The Colored Petri Nets modeling language

In this section, we outline the formal semantics of Colored Petri Nets (CP-nets) as they are defined in [6].

Definition 2.1 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 $\{m(s) \mid s \in S\}$ are the coefficients of the multi-set.

Definition 2.2 A 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:
$\forall t\in T$: $Type(G(t))=B \wedge Type(Var(G(t)))\subseteq\Sigma$
(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:
$\forall a\in A$: $Type(E(a))=C(p)_{MS} \wedge Type(Var(E(a)))\subseteq\Sigma$, with $p=N(a)$
(ix) I is an *initialization function* that maps each place $p\in P$ into a closed expression of type $C(p)_{MS}$:
$\forall p\in P$: $Type(I(p))=C(p)_{MS}$
When we draw a CP-net we omit initialization expressions, which evaluate to $\emptyset$.

By convention, we write the names of the places inside the ellipses. Each place has an associated *data type* (color set) determining the kind of data, which the place may contain. The type information is written in italics, next to the place. 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. The types of a CP-net can be arbitrarily complex, e.g., a record where one field is a real, another a text string and a third a list of integers.

The actions of a CP-net are represented by means of transitions. 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 the arc expressions, which are positioned next to the arcs. Arc expressions may contain variables as well as constants.

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

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

To talk about the *occurrence of a transition*, we need to bind incoming expressions to values from their corresponding types.

Definition 2.3 A binding of a transition t is a function b defined on Var(t), such that:
(i) $\forall v\in Var(t)$: $b(v)\in Type(v)$
(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 $G(t)\langle b\rangle$ - results in true.

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

From the forenamed definitions we see that it is possible to attach a boolean expression with variables to each transition. The boolean expression is called a guard and specifies that we only accept bindings for which the boolean expression evaluates to true.

Definition 2.4 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 2.5 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.

CP-nets can be analyzed, either by means of simulation, formal analysis based on the construction of occurrence graphs (representing all reachable markings), calculation and interpretation of system invariants (called place and transition invariants) and the check of structural properties, which guarantee certain behavioral properties.

In CPN Tools, 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. We can use 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 CP-nets specified in a number of hierarchically related pages. The companion state space tool generates the entire or a portion of the model's state space and a set of standard as well as user defined analysis queries.

3 The information flow security CP-net

In the OMG CORBA Security setting, access policies are defined based on *privilege* and *control attributes* and access decisions are made via a standard access decision interface. The *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. 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, but does not prescribe specific policy composition rules.

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 table-based representation of a sample access policy. As subject entries we use the privilege attributes possessed by the principals. 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 of a `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 membershiphs 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 policy introduced in Tables 1 and 2.

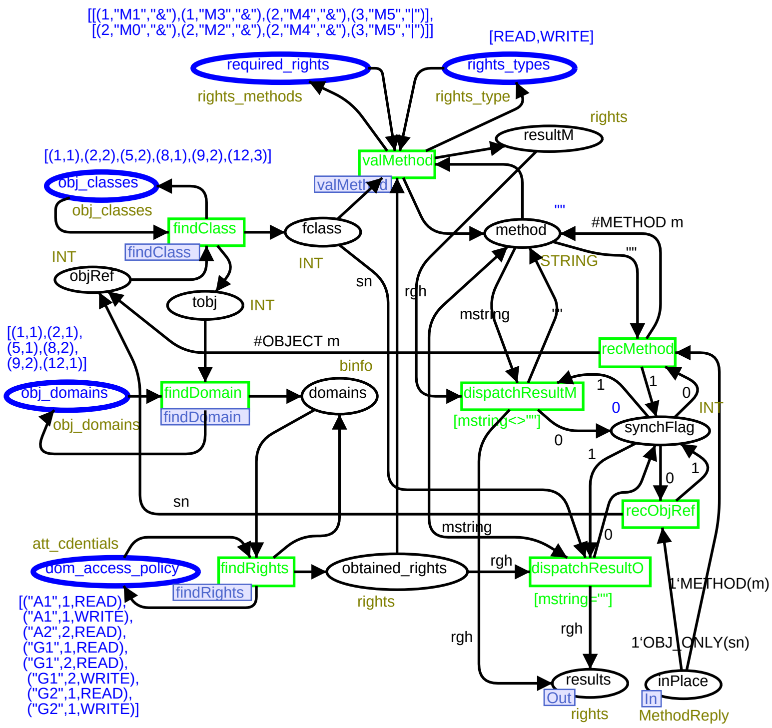

Fig. 1. The access decision CP-net submodel

The `AccessDecision` object determines the validity of invocation requests relying on the privilege and control attributes provided by the `AccessPolicy` and `RequiredRights` objects. CORBA Security does not prescribe how an `AccessPolicy` object combines rights granted by different privilege attribute entries and this allows for potentially unlimited flexibility in security policy specification. Figure 1 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 is specified (bold place `dom_access_policy`) 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`.

The CP-net of Figure 1 corresponds to the `protSys` substitution transition of the CP-net shown in Figure 2 that mimics a method execution: an object sends one or more messages (specified at the bold input/output place `obj`) to itself or to other objects. Access to an 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 object identifiers (color `binfo` for the place `AOsL`) for all objects in which read operations (allowed by the used access control) were performed. This list is termed as *Accessed Objects List* and it is repeatedly transmitted backward (replies) and forward (requests) as prescribed by the method's message sequence specification.

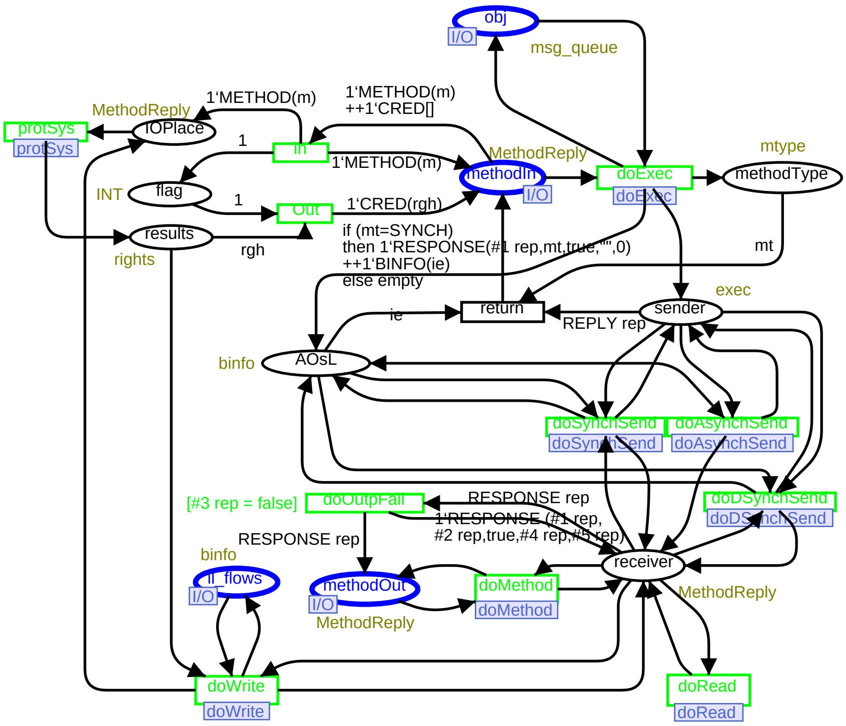

Fig. 2. The top layer of the method execution CP-net

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 `x` (source) to an object `y` (target) *is not secure*, 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 *y 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 transmitted via the AOsL list.

Figure 3 summarizes the color, variable and function declarations used for the transition and arc inscriptions of the CP-nets of Figures 2 and 4.

```
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. 3. Colors, variables and functions used in CP-net incriptions

Due to space limitations we omit the details of the `doSynchSend`, `doAsynchSend` amd `doDSynchSend` substitution transitions shown in Figure 2. We note that the AOsL list is never changed as a result of an asynchronous method execution. In the two other cases, the method reply is returned together with a list of object identifiers for all objects, in which read operations were allowed. The AOsL list is subsequently updated by calculating the union with the existing list of accessed objects (function `unio`).

A system model is composed of a number of interacting instances of the CP-net shown in Figure 2 and each model instance represents a particular method execution. For all method executions their instance input places (`obj`, `methodIn`) are updated as prescribed by the system's object method dependencies. Insecure information flows are detected at the `doWrite` substitution transition and for each object are separately recorded at the `il_flows` output places.

Figure 4 reports the simulation results given for the shown case system model and the access control of Figure 1. Insecure information leakage has been detected and recorded in the `il_flow_obj2` and `il_flow_obj9` places. We observe the existence of insecure information flows from `o1` and `o5` to `o9` and from `o8` to `o2`.

The simulated model possesses the validity of a formal verification process, with respect to the detection of existing insecure information flow paths.

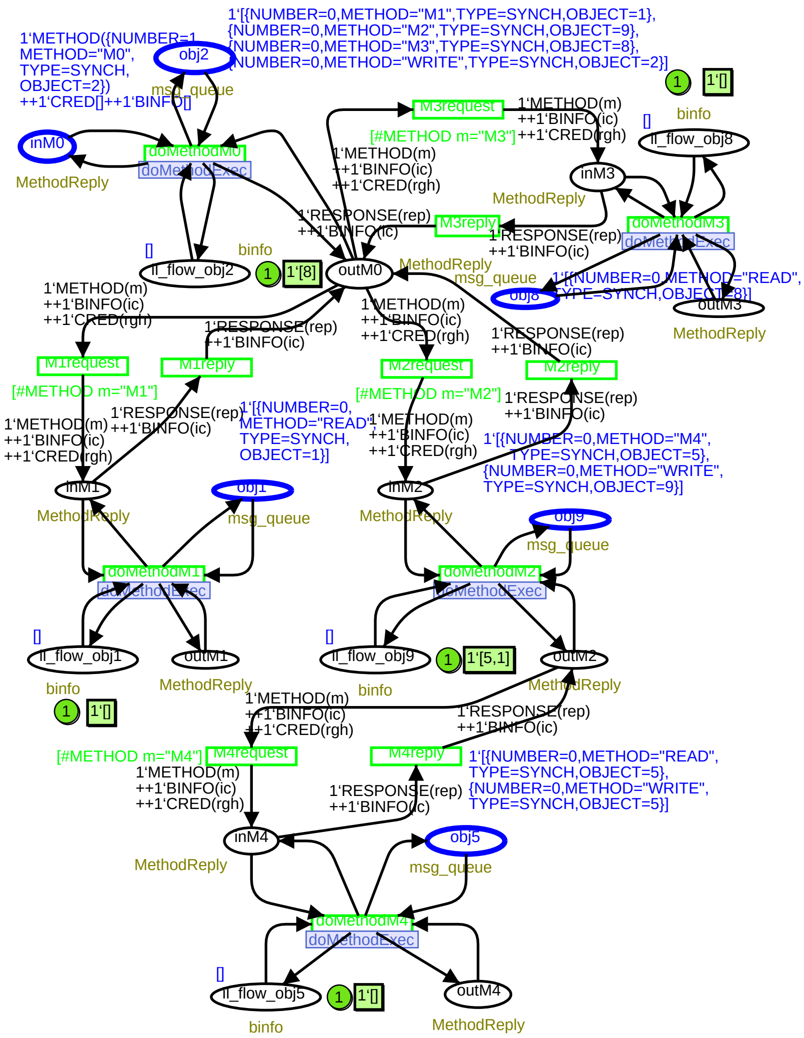

Fig. 4. A case system model and the resulted insecure information flow paths

4 State space analysis

Analysis of the occurred information flows 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 ([5]). Given the full state space, also known as occurrence or reachability graph, we are able to check a set of standard properties (bounds-related, home, liveness and fairness properties), as well as the existence of an occurrence sequence (reachability) to a particular marking (state).

Figure 5 summarizes the results for the standard checks performed in CPN Tools. The full state space is generated in 13 secs and consists of 2029 nodes (states) and 3282 arcs. As expected, there is a single dead marking (node number: 2029) that provides us the required information, with respect to the occurred insecure flows (see Figure 4).

```
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. 5. The state space analysis standard report for the CP-net of Figure 4

Table 4 summarizes the information flow data derived by exploiting a set of predefined ML functions to explore the generated state space.

The results obtained for query 1 verify the simulation results shown in Figure 4 regarding the insecure flows detected at the found dead marking. In query 2, we use the function `SearchNodes` to search the state space for a marking that yields all flows (including the secure ones) to `o2`. Queries 3 and 4 reveal the details of the insecure flow (definition 3.1) sourced at `o8`.

Function `SearchNodes` provides us unlimited flexibility in the specification of flow data deriving queries. 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 proposed analysis alternatives our model fits to the modular state space analysis described in [3]. Unfortunately, CPN Tools does not currently support the generation of separate state space modules and the application of the forenamed analysis approach remains an open prospect.

Table 4. Non-standard state space queries

1. Insecure information flows:		
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 flows to o_2:		
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

5 Related work

Information flow security is an active research problem that was first approached in 1973 ([8]) and that is still attracting the interest of researchers in a number of recently published works ([10], [2], [4]). Recent works in the context of distributed objects ([4], [13], [12]),

- are based on different and often not realistic assumptions on when an information flow occurs,
- do not always take into account that 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.

The present work (i) takes into account the bi-directional nature in the direct or indirect information transfer between senders and receivers, (ii) allows for modeling nested object invocations, (iii) employs the CORBA Security reference model and thus it is not bound to a specific access control model and (iv) does not assume proprietary run-time support.

6 Conclusion

In modern networked business information systems confidentiality cannot be assured by controlling access to objects without taking into account the information flow paths implied by a given, outstanding collection of access rights.

In this work, we proposed a modeling approach that possesses the virtue of simulating insecure information leakage in a graphical environment and allows querying the model about the detected information flow paths and their dependencies. The studied access control is specified as prescribed by the standardized OMG CORBA Security service. Our model provides a view of the detected information flow paths and in this way supports the design of mandatory access control, where we are interested to specify and enforce an appropriate set of object classification constraints to prevent undesirable leakage of sensitive information.

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. Beznosov, K., Deng, Y.: A framework for implementing role-based access control using CORBA security service, In: Proceedings of the Fourth ACM Workshop on Role-Based Access Control, Virginia, USA, 1999, 19-30
2. 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
3. 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
4. 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
5. Jensen, K.: An introduction to the practical use of colored Petri Nets, In: Lectures on Petri Nets II: Applications, LNCS 1492, 1998, 237-292
6. Jensen, K.: An introduction to the theoretical aspects of colored Petri Nets, In: A Decade of Concurrency, LNCS 803, 1994, 230-272
7. Karjoth, G.: Authorization in CORBA security, In: ESORICS'98, LNCS 1485, 1998, 143-158
8. Lampson, B. W.: A note on the confinement problem, Communication of the ACM, 16, 10, 1973, 613-615
9. Larsen L., Harrold, M.: Slicing object oriented software, In: Proceedings of the 18th International Conference on Software Engineering, 1996, 495–505
10. 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
11. Object Management Group: Security service specification, version 1.7, OMG Document 99-12-02, 1999
12. 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
13. 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