A History-Dependent Access Control Mechanism Using Reflection

Massimo Ancona[1], Walter Cazzola[2], and Eduardo B. Fernandez[3]

[1] DISI - Department of Informatics and Computer Science,
University of Genova, Genova, Italy
ancona@disi.unige.it

[2] DISCO - Department of Informatics, Systems, and Communication,
University of Milano - Bicocca', Milano, Italy
cazzola@dsi.unimi.it

[3] Department of Computer Science and Engineering,
Florida Atlantic University, Boca Raton, FL, USA
ed@cse.fau.edu

Abstract

We propose here a mechanism for history-dependent access control for a distributed object-oriented system, implemented using reflection. In a history-dependent access control system, access is decided based not only on the current request, but also on the previous history of accesses to some entity or service. We consider timing constraints expressed using temporal logic, and we describe a possible implementation for our mechanism. The expected benefits from the reflective approach are: more stability of the security layer (i.e., with a more limited number of hidden bugs), better software modularity, more reusability, and the possibility to adapt the security module with relatively few changes to other applications and other authorisation policies.
Keywords: Access Control Mechanisms, Authorisation, History-Dependent Constraints, Object-Oriented Systems, Reflection, Security

1 Introduction

Security implies not only protection from external intrusions but also controlling the actions of internally-executing entities and the operations of the whole software system. The interleaving of operations and data protection actions may become very complicated and often intractable. For this reason, security must be specified and designed in a system from its early design steps [10].
From another point of view

- it is very important that the security mechanisms of the application be correct and stable;
- the security code should not be mixed with the application code, otherwise it should be very hard to reuse well-proven implementations of the security model.

If this is not done, when a new secure application is developed the designer|implementer wastes time to re-implement and to test the security modules of the application. Moreover, security is related to: “who is allowed to do what, where and when”; so security is not functionally part of the solution of the application problem, but an added feature defining constraints on object interactions. From this we can think of security as a feature operating at a different computational level and we can separate its implementation from the application implementation.
In previous works [2, 3] we have shown that exploiting some typical reflection features, such as *separation of concerns* and *transparency*, it is possible to split a secure system into two levels: at the first level there are (distributed) objects cooperating to solve the system application; at the second level, rights and authorisations for such entities are identified, specified and mapped onto reflective entities which transparently monitor the objects of the first level and authorise access to other objects, services, or information. In this way it is possible to develop stable and reliable abstractions for handling security. It is also possible to reuse them during system development, thus reducing development time and costs, and increasing application level assurance.

This work attempts to prove the flexibility of the reflective approach, carrying over the concepts and ideas introduced to design a system implementing a role-based access control policy, to a more complex policy, history-dependent access control.
History-dependent access mechanisms validate access requests based on previous system computations [11]. Those mechanisms are more complicated than the mechanisms for role-based access control, because they use dynamic information (the history) in order to validate access requests. In spite of their complexity they permit to express

dependencies involving several different execution instants. For example, we can express constraints such as: a user can read some information only if before he has received the permission to read it from the owner, or an object can use a service only if it had never used it before. Similar constraints are hard to express in role-based terms.

The paper is organised as follows: in Section 2 some preliminary concepts related to computational reflection are presented; in Section 3 we describe a model for history-dependent constraints; in Sections 4 and 5 we present more details of the model and some ideas on how to implement it. In Sections 6 we prove the flexibility of our approach modelling the Chinese Wall policy [5]. In Section 7 we analyse some related works and we describe the possible evolution of this work. We conclude with some remarks on the reflective approach and the benefits and drawbacks it involves for security enforcement.

2 Preliminary Concepts

2.1 Authorisation Rules and Access Control Mechanisms

An authorisation system plays a monitoring role, judging if the requests sent by an object to another object are permissible requests. The judgement uses security information based on *objects* and *subjects*; where a subject represents an entity performing or requesting an activity (i.e., an active object playing the client role), while an object is a passive entity supplying a service (i.e., a passive object or an active object playing the role of server). Authorisation constraints may be modelled using the access matrix model [12]. In this model the authorisation rules are described by a bidimensional matrix indexed on subject and objects and access of type t to an object o is allowed to a subject s when the entry $\langle \mathsf{s},\mathsf{o} \rangle$ of the matrix contains access type t. Such a model can be realized by *capability lists*, *access control lists*, or combinations of these. Examples of constraints suitable to be modelled by an access matrix are the access modes of the Unix file system, or the Oracle authorisation system.

The access matrix expresses authorisation constraints independent from previous interactions involving other entities; we call such constraints *instantaneous*. Obviously by using that model it is impossible to express constraints whose evaluation involves information about the previously allowed services. We call such constraints *history-dependent*. An example of a history-dependent constraint is: an object o_1 can access a reserved document d handled by another object o_2 only if o_1 has received the authorisation to read the document by its author, the object o_2. In the literature, similar situations are modelled by using history-dependent access control mechanisms (see for example [11, 4]).

2.2 Background on Reflection

Computational reflection, or just reflection, is defined as the activity performed by an agent when doing computations about itself [13]. A reflective system is logically structured into two or more levels, which constitute a *reflective tower*. Entities working in the base level, called base-entities or reflective entities, define the basic system behaviour. Entities working in the other levels (meta-levels), called meta-entities, perform the reflective actions and define further characteristics beyond the application-dependent system behaviour. Each level is causally connected to adjacent levels, i.e., entities belonging to a level maintain data structures representing (or, in reflection parlance, reifying) the states and the structures of the entities in the level below. Any change in the state or structure of an entity is reflected in the data structures reifying it, and any modification to such data structures affects the entity's state, structure and behaviour. Computational reflection allows adding properties and functionality to the application in a transparent manner (separation of concerns) [18]. For a classification and comparison of classic approaches to reflection see [6].

2.3 Channel Reification Model

This model [1] is an extension of the message reification model, aimed to overcome some of its limitations, while keeping its advantages. Channel reification is based on the following idea: a method call is considered as a message sent through a logical channel established between an object requiring a service, and another object providing such a service. This logical channel is reified into an object called *channel* (as shown in figure 1). A channel is characterised by a triple composed by the objects it connects and by the kind of meta-computation it performs.

$$\textbf{channel} \equiv \textbf{(client, server, channel\_kind)}$$

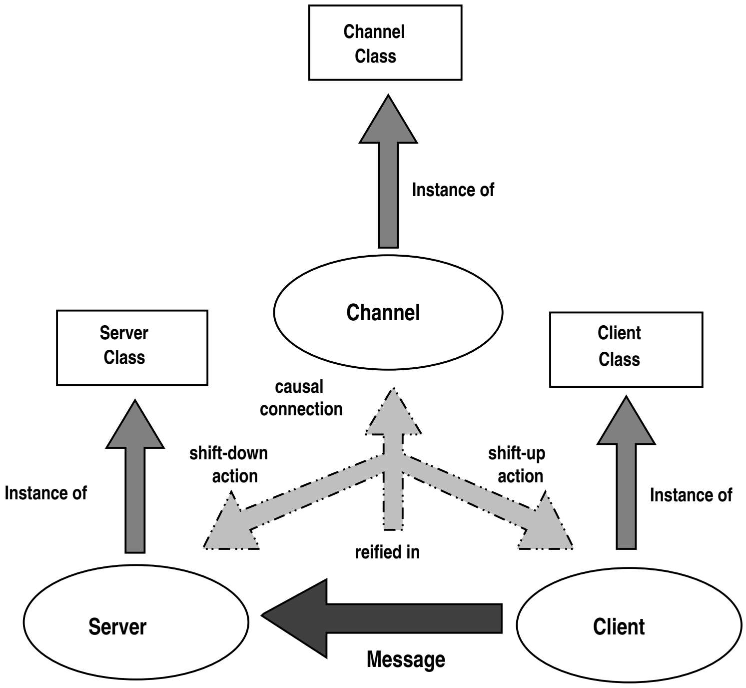

Figure 1: **Channel reification model scheme**

A *channel kind* identifies the meta-behaviour provided by the channel. In a typed object-oriented language the kind is also the type of the channel class. The kind is used to distinguish the reflective activity to be performed: several channels (distinguishable by the kind) can be established between the same pair of objects at the same moment.
The lack of information continuity of message reification is eliminated by making channels persist after each meta-computation. A channel is reused when a communication characterised by the same triple is generated. In this way, meta-entities are created only once (when they are activated for the first time), and reused whenever possible. When an object is destroyed, all channels established from|to it are also destroyed. This lifecycle limits channel proliferation, since a garbage collector erases dangling channels.

Each service request is trapped (shift-up action) by the channel of the specified kind connecting client and server objects (if it exists); otherwise, such a channel is created. In either case, the channel then performs its meta-computation and transmits the service request to the supplier. The server's answer is collected and returned to the requesting object (shift-down action).

3 History-Dependent Constraints

We concentrate here on validation of method calls (also termed service requests) from one object to another. With this restriction we do not lose generality, because each access request can be interpreted as a method call. We use the symbols

$$\mathsf{o}_1 \blacktriangleright \mathsf{o}_2[\,\mathsf{m}(\,\mathsf{args}\,)\,]$$

to indicate a request (both before and after the validation phase depending on the context) from the object o_1 to the object o_2 for the method m with arguments args. A generic request is indicated by δ_i and the set of all possible requests is Δ.

Definition 3.1 (History of system interactions at time i)
The history of system interactions at time i is the sequence of all services authorised before time i. The history sequence is partially ordered by the occurrence relation.

We adopt a *history* concept similar to the one in [9], tied to *event occurrences*, and not related to absolute *time*. Each event $\delta_i \equiv \mathsf{o}_1 \blacktriangleright \mathsf{o}_2[\,\mathsf{m}(\,\mathsf{args}\,)\,]$ is an authorised service. From the definition 3.1 a history $\hbar$ can be seen as a path of consequent events (δ_i)

$$\hbar \equiv \sigma_0 \xrightarrow{\delta_1} \sigma_1 \xrightarrow{\delta_2} \sigma_2 \cdots \sigma_{k-1} \xrightarrow{\delta_k} \sigma_k$$

which leads the system from a state (e.g., σ_0) into another (e.g., σ_k); all the other σ_i are intermediate states. From the access control point of view the only significant information are the allowed services and not the intermediate states. Of course, the paths leading to a certain state are not unique and they depend on the order in which events occur. The interactions of a system are then modelled as a set of paths, i.e., a graph, whose paths are equivalent to the ones considered in branching temporal logic [17].

Due to the similitude with the branching temporal logic model, the constraints for the authorisation of a service can be expressed using the connectives of that logic; in particular only a subset of them is necessary, containing connectives related to the past (i.e., so far $\Box_{\mathrm{p}}$, once $\Diamond_{\mathrm{p}}$, since $\mathcal{S}$, before $\circ_{\mathrm{p}}$) and connectives that bind a formula to a path ($\triangledown$, and $\triangle$). Thus, each constraint for the authorisation of the service δ is expressed by a formula Γ, and we denote it by

$$\Gamma \Rightarrow \delta$$

which means that δ is allowed only if Γ is satisfied.

Due to the nature of the constraints we need to express, all the formulas are *anchored* to the current state σ_0 of the system, and have to be satisfied for all branches leading to the current state σ_0; thus each generic formula is written

$$\triangle(\sigma_0, \Gamma) \Rightarrow \delta.$$

In what follows, for conciseness we omit the anchor and the constraints simply become $\Gamma \Rightarrow \delta$.

Instantaneous constraints are special cases of history-dependent constraints and in order to use a unique formalism they are expressed using formulas with always true premises:

$$\texttt{true} \Rightarrow \delta.$$

It is important to note that the set of paths includes|describes the whole system history from the point of view of the allowed services. In this way, it is simple to add, remove or modify constraints. History contents does not depend on what constraints have to evaluate, to forbid or to monitor.

In appendix A we explain in more detail history and constraint semantics.

3.1 Example of Use of Temporal Constraints

The authorisation system for this example considers a bank and its ATM network, so the objects involved are: clients, banks, ATMs, and bank employees (bank VPs, accountants, and other employees).

Supply on Request

Only bank account holders can draw from the bank ATM. This fact can be expressed as: the client can obtain money only if is an account holder in that bank and asks to draw money from the ATM.

$$\circ_{\mathrm{p}} \langle \lambda x.x = \mathsf{Client} \blacktriangleright \mathsf{ATM}[\,\mathsf{drawing}(\,\mathsf{sum}\,)\,] \wedge \mathsf{isAccountHolder}(\,\mathsf{Bank},\,\mathsf{Client}\,)\rangle \Rightarrow \mathsf{ATM} \blacktriangleright \mathsf{Bank}[\,\mathsf{drawing}(\,\mathsf{Client},\,\mathsf{sum}\,)\,]$$

for all clients, and amounts. Of course we also have to check if his account has money before allowing the drawing; in this example we do not consider this aspect because it does not add new elements to the explanation.

Resource Lock

An account holder cannot draw more than \$1,000 within a month. As soon as he exceeds that limit no further drawings are possible.

$$\neg\langle \lambda x.x = (\,\mathsf{Client} \blacktriangleright \mathsf{ATM}[\,\mathsf{drawing}(\,\mathsf{p}_1\,)\,] \wedge (\,\mathsf{p}_1 > \$1{,}000\,)\,)\rangle\ \mathcal{S}\ \langle \lambda x.x = \mathsf{ATM} \blacktriangleright \mathsf{ATM}[\,\mathsf{month\_end}\,]\rangle \Rightarrow \mathsf{Client} \blacktriangleright \mathsf{ATM}[\,\mathsf{drawing}(\,\mathsf{p}\,)\,]$$

for all clients and amounts p, and p_1.

The above constraint means (Fig. 2) that the client, in order to obtain some amount of money must not have exceeded the \$1,000 limit since the end of last month.

Avoiding Information Smuggling

In a bank, some roles, e.g., accountants, receive private information such as salaries, from different employees. In this case, information about a person should not be disclosed to other persons, e.g., the salary of the bank VP should not be shown to other employees.

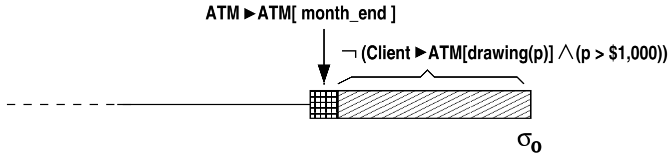

Figure 2: **Explanation for constraint**

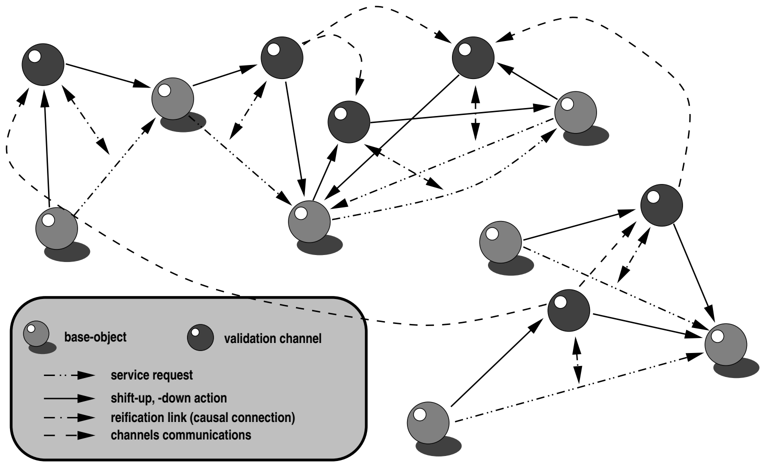

Figure 3: **Reflective history-based access control system**

$$\Box_{\mathsf{p}} \neg(\exists \mathsf{BVP}{\in}\mathsf{Bank\ VP}.\langle \lambda x.x = \mathsf{BVP} \blacktriangleright \mathsf{Accountant}[\,\mathsf{inform}(\,\mathsf{p}\,)\,] \wedge \mathsf{p} = \mathsf{p}_2\,\rangle\,) \Rightarrow \mathsf{Accountant} \blacktriangleright \mathsf{Employee}[\,\mathsf{inform}(\,\mathsf{p}_2\,)\,]$$

for all bank VPs, employees, and accountants.

Note that in some examples we use predicates, such as $>$ and $=$, expressed in their infix form. These examples show only a few of the possible situations of interest but give an idea of the application of this policy.

4 The Reflective Approach

How to use reflective features in order to model role-based access control mechanisms have been studied in [2, 3, 14]. Reflection allows the separation of nonfunctional from functional code, organizing the access control policy in a specific meta-level connected to the rest of the application by the causal connection relation. In general, access control is enforced at the object communication level; for this reason communications-oriented reflective models (such as the channel reification approach) are more suitable to model them than meta-object oriented models [6]. The main advantages of using a communications-oriented reflective approach is represented by the fact that a malicious request cannot reach the server, because it is trapped by the meta-entity, which validates the request.
Those results has been determined for role-based access control mechanisms, but they can be also applied to history-based access control mechanisms.
Modeling a history-based control policy using the channel reification model is based on associating a channel to each communication between two objects (as in Fig. 3). Channels trap each request before it reaches the called object (see Section 5.1 to get a better understanding of the shift-up mechanism). Once the action is trapped, the channel looks up the system constraints and the history in order to validate the request. When the request is validated, the channel either signals to the caller that the service is forbidden or lets the callee perform the request (see the sequence diagram of Fig. 4 for more details of the validation phase; the diagram is based on the bank example: `supply on request` and the service is allowed). The history is built from allowed requests, and each channel contributes to its construction with their set of allowed requests. We can keep system history in two ways: centralized or decentralized. In a large distributed system it is hard and inefficient to keep the history centralized, because it is necessary to charge an entity (history holder) with this task. The history holder has to communicate with all channels in order to collect all history fragments and in order to supply them histories, that they will use to

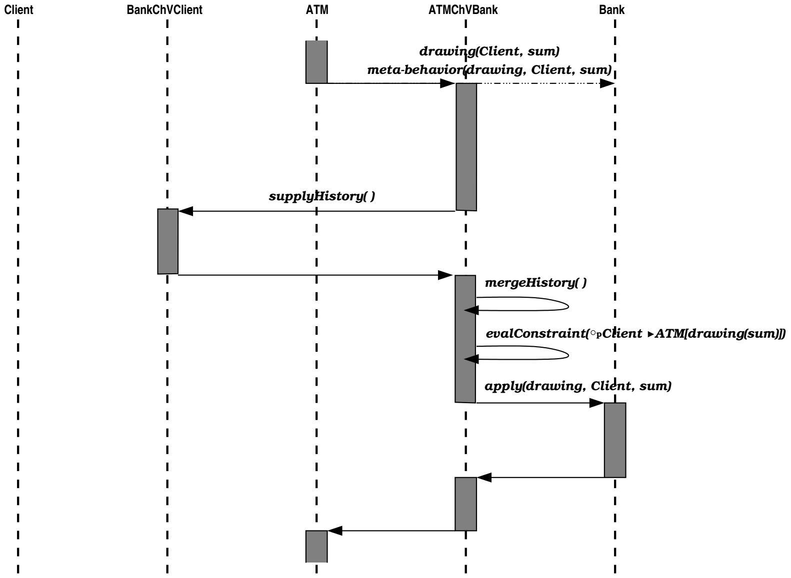

Figure 4: **Sequence diagram for the validation mechanism**

evaluate the constraints before allowing a service to be accessed. This behavior increases the overhead due to the objects' communication, but it guarantees history consistence. History decentralization is achieved by charging each channel with the task of holding the part of history managed by it. In this way the bottleneck represented by a centralized history holder is avoided, but when a channel has to evaluate a history-based constraint, it has to complete the information it holds with the part of history, necessary in the evaluation, held by other channels, so there are communications only when needed and rarely with the same entity.

5 Implementation

We are currently developing a Java prototype for this mechanism, where we use implementation ideas from an earlier prototype [3] (available from `http://www.disi.unige.it/person/CazzolaW/OORSecurity.html`). In this section we present channel class ValidationChannel, which is the kernel of the mechanism which realizes the history-based access validation policy.
In the case of reflective history-dependent access control we have only one kind of channel: ValidationChannel. Channels of that kind are established between two object and they check if any request exchanged between their referents can be allowed. Two phases can be identified in the execution of ValidationChannel:

- *initialization*: at channel bootstrap, the channel loads the constraints related to the services offered by its referent (the one playing the role of server), merging together the constraints related to a unique request (remembering that a service can be allowed only if all its constraints are satisfied, and if $(\Gamma_1 \Rightarrow \delta) \wedge (\Gamma_2 \Rightarrow \delta) \equiv \Gamma_1 \wedge \Gamma_2 \Rightarrow \delta$ we substitute each group of constraints related to a unique request with a formula which joins all the premises);

- *validation*: each channel spends most of its lifecycle waiting for a request and validating it; each request is trapped during the shift-up action (Section 5.1), and the meta-behavior method handles the constraints validation;

Each channel has a method called meta-behavior which is automatically invoked during the shift-up. In the case of the ValidationChannel it performs the following steps:

- the first step consists of reassembling the historical environment in which the constraint will be evaluated:
  - it consults its constraint database (built during the initialization phase) looking for the constraints related to the request it must validate;
  - once it retrieves the constraints to evaluate, it scans them in order to determine the previously requests because it must know if they are allowed or not;

    - from the previously retrieved information it goes back to those channels that have the missing part of the history, needed for the evaluation;
    - it asks the involved channels the missing part of the history, and it reassembles all histories into a unique history (see Section 5.2);

- when it has rebuilt the historical environment it begins the evaluation of constraints (see Section 5.3);
- when it has evaluated the constraints, if the request must be rejected, the channel indicates this to the client, otherwise it stores the request in the history as 'allowed'.

In the following we explain in more detail some aspects of the validation: how the reflective behavior is realized, history management, and the constraints evaluation routine.

5.1 Shift-Up, -Down Mechanisms

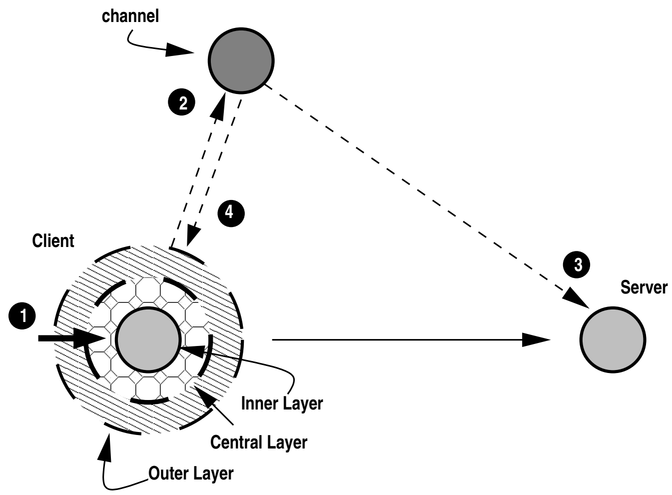

Figure 5: **Shift-up, and -down mechanism**

Due to the *global view* property (see [6]) of the channel reification model, each reflective action starts from the client and involves a client, a server, and a channel in its execution. The reflective behavior is achieved by overriding the normal behavior of the binding|look-up between the client and the server, and the remote method invocation in order to divert the request from the server to the channel. The client is composed of three layers (see Fig. 5). The inner layer supplies the operations necessary to connect and to communicate with the server in a normal context. The central layer encapsulates all operations of the inner layer and uses them to connect and to communicate with the right channel, realizing the shift-up,-down actions. The outer layer defines the behavior of the client.
The server code remains unchanged. There is no need to wrap it because the server receives requests that have been previously filtered by the channels, it cannot be accessed directly by the clients. When a client issues a request, the control is dispatched to the central layer of the client (step ❶ in Fig. 5) which determines the right Validation channel to be invoked on the basis of the identity of the objects involved in the request. The request is then forwarded to the channel while the client idles until a reply is sent back to it (step ❷ in Fig. 5). The channel validates the request and, if legal, forwards it to the server (step ❸ in Fig. 5). When the channel receives the reply from the server, it forwards it back to the client (step ❹ in Fig. 5). The server is unaware of the validation process: it executes only filtered requests, achieving in this way a good *separation of concerns* and implementation modularity.

5.2 History: Data Structures and Management

We chose to decentralize the history, entrusting each channel with the management of its own history fragment. In this way we reduce the overhead due to unnecessary communications, although it makes more complex to determine the historical environment in which to evaluate each constraint (see Section 5.3 for more details).
From definition 3.1, a history is an ordered sequence of allowed requests. An allowed request is a pair (δ, i), where δ is the request and i is the time in which that request was allowed. Time is an integer count increased at

any allowed request. Of course, the time count in the system has to be unique and increase consistently with the order in which the requests have been allowed, but it is not a problem to build a server offering this service to the channels. Each history fragment is a list of allowed requests, ordered with respect to the time index, and it is stored into a channel attribute. History management consists of two actions: the first (addToHistory()), simply adds an allowed request to the history kept by the channel, the second (mergeHistory()) merges two histories, one passed as parameter to the one kept by the channel, keeping the event order. The merge routine is a normal merging of two ordered lists and can be performed, in the worst case, in linear-time.

Besides these actions which manipulate directly the history fragment kept by a channel, we also need a method (supplyHistory()) which returns the history fragment kept by the channel invoking this method. This method is remotely invoked by another channel when it tries to complete the historical environment, before starting the constraint evaluation. An improvement to the history management routines consists of keeping an incremental history. Each channel keeps its history fragment merged together with the history fragments obtained from other channels and related to entities different from its referents, and when it needs to complete the historical environment it asks the other channels only an updating of its information and not their complete history. In this way we speed-up the merging routine and, above all, we reduce the communication time needed to receive the complete history kept by another channel.

5.3 The `evalConstraint` Function

In spite of Java support of unicode we prefer to express constraints without using special symbols. Thus $\langle \lambda x.\Gamma(x) \rangle$ becomes `onlabel x` Γ`(x)`, $\Box_p \Gamma$ becomes `sofar` Γ, $\circ_p \Gamma$ becomes `before` Γ, $\Diamond_p \Gamma$ becomes `once` Γ, $\neg\Gamma$ becomes `not` Γ, and so on. As an example, the formula presented earlier becomes:

`sofar not exists BVP in Bank VP (onlabel x (x=BVP▶Accountant[inform(p1)] and (p=p1)))` (5.3.1)

Evaluation of constraints is the main function of the ValidationChannel, its routine (`evalConstraint`) depends on how the constraints are coded. We chose to encode each constraint as a binary tree. Each tree node is a connective and its children are the connective operands. Figure 6 shows the tree obtained from the constraint above. It should be noted that the leaves of the tree are labeled by predicates or by service requests (each request is denoted by the symbol ▶).

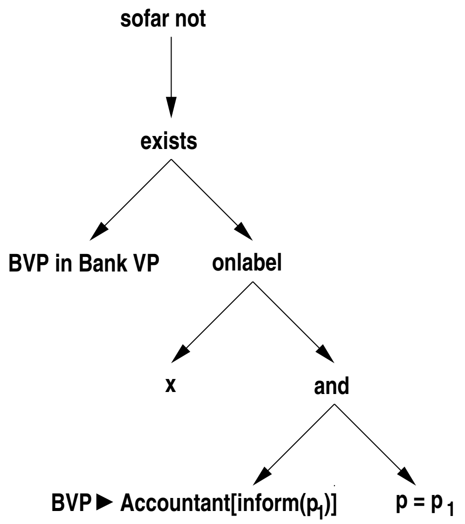

Figure 6: **Constraint tree for formula 5.3.1**

The evaluation consists of traversing the tree for evaluating subexpressions, and associating with the root a boolean value depending on the values of its subtrees. The evaluation of first-order logic connectives, such as `and`, and `or`, is simple and well-known in the literature, thus we do not discuss it. More complex and less known is the evaluation of the temporal connectives. Obviously, temporal connectives, in order to be evaluated, need a temporal anchor. The temporal anchor is an integer representing the time at which we perform the evaluation. The first step performed by `evalConstraint` is to retrieve the current time and to use it as an anchor. The `evalConstraint` applied to service requests checks if the request belongs to the history, i.e., if it occurred at the time specified by

time	BVPDonChAccJane	BVPDonnaChAccJane
1		BVPDonna ▶ AccJane[inform(`"John salary is $500"`)]
5		BVPDonna ▶ AccJane[inform(`"Donna salary is $1,200"`)]
7	BVPDon ▶ AccJane[promote(`"Joe"`)]	
12		BVPDonna ▶ AccJane[public_inform(`"deduction on John salary 20%"`)]
20	BVPDon ▶ AccJane[inform(`"Joe salary is $800"`)]	
22	BVPDon ▶ AccJane[public_inform(`"deduction on Joe salary 20%"`)]	

Table 1: History example

the anchor. In order to handle the evaluation of nested temporal subformulas we detach `evalConstraint` from the current time, but we make it parametric on the temporal anchor. In this way it is possible to recursively call `evalConstraint` in order to evaluate the constraint subformulas simply updating the anchor in agreement with the kind of the connective and with the fact that each subformula has to be true in a moment previous or equal to the one in which the formula has to be true.
In order to simplify the evaluation algorithm we have analyzed the constraint typology, and from that analysis we determined that all the history-dependent constraints we consider express only punctual temporal properties (as in the step before event φ has occurred, or once event φ has occurred), or continuously negative properties (as so far event φ never occurred, or event φ_2 never occurred since event φ_1 occurred). Thus it is possible to merge `sofar`, and `since` evaluation with the corresponding `not` connective evaluation making easier their evaluation simply looking for the subformula occurrence of the other connectives.
Predicates and `onlabel`, `onstate` connectives evaluation is directly linked to the system history and state.
The evaluation of all temporal connectives consists of modifying the temporal anchor in agreement with the connective kind, and to apply `evalConstraint` to the connective operands with the new temporal anchor. We do not apply the evaluation when we reach a leaf of the tree, in that case we have to consult the history. Thus in the `before` case (the simplest) we delay by one the temporal anchor and we evaluate its operand; in the `sofar not`, and `once` cases we loop on the operand evaluation delaying by one the temporal anchor until the operand evaluation is true (in the case of `sofar not` the routine returns false instead of true). In the `since not` case we delay by one the temporal anchor and we loop on the second operand until the evaluation becomes true. In this way a new temporal anchor is determined and after we evaluate the first operand on the interval delimited by the new temporal anchor and the initial temporal anchor in order to check if it is always false. In that case the `since` evaluation is true otherwise it is false.
Now, we explain the `evalConstraint` by an example. Accountants play a very particular role. There is information that they share with bank VPs, but that they cannot share with the other employees (such as the employees' salary). In order to avoid information smuggling through the accountant, the validation system has to verify the source of each information which the accountant passes to each employee. The constraint (or better the constraint family) implementing this behavior is represented by formula 5.3.1. Let us consider the situation in which there are an accountant (Jane), two bank VPs (Don, and Donna), and an employee (Joe), and the history kept by each channel (the one established between the two bank VPs and the accountant, BVPDonChAccJane, and BVPDonnaChAccJane respectively) is shown in table 1. When Jane wants to communicate with Joe, e.g, by performing the request AccJane ▶ EmpJoe[inform(`"your salary is $800"`)], the validation system, in order to validate the request, has to evaluate formula 5.3.1. The historical environment in which AccJaneChEmpJoe evaluates the constraint is represented by the fusion of the second and the third columns of table 1 and the current time is 25. The `evalConstraint` starts evaluating the connectives `sofar not`; its evaluation consists of searching an instant in which a bank VP exists which satisfies `onlabel x (x=`BVP ▶ Accountant[inform(p_1)] `and (p=`p_1`)`. It starts to search from 25, and delays by one until it reaches time 20, which satisfies the subformula BVP ▶ Accountant[inform(p_1)]. Then it tries to match[1] the information Jane obtained from Don with the information Jane wants to pass to Joe, failing. Then it continues to search until reaching time 1. Thus that operand is not matched and Jane can pass that information to Joe. In the case when the information source was a bank VP, one of the matches shouldn't fail, and the channel would communicate to Jane the rejection of her request.

6 Modeling Chinese Walls

The *Chinese Wall* policy [5] arises in the financial segment of the commercial sector which provides consulting services to other companies. The Chinese Wall is a history-dependent policy whose objective is to prevent in-

[1] In this example we handle only a simple way of smuggling information through exact match, it is possible to use more complex matchings through predicates.

formation flows which cause conflict of interest for individual consultants. In this section we briefly present the Chinese Wall policy and show how to model its behavior using temporal logic constraints.
We have information about different entities (e.g., about rival companies) which can be accessed only in mutually exclusive manner from any subject. The Chinese Wall policy categorizes such entities into mutually disjoint *conflict of interest* classes. Entities in conflict belong to the same class, and each subject can only access information about a single entity per each conflict of interest class. As stressed in [15] the Brewer-Nash model this behavior can be expressed by the following mandatory rules.

❶ *Brewer-Nash Read Rule*: subject S can read object O only if

- O belongs to the same entity as some objects previously read by S (i.e., O is within the wall), or
- O belongs to a class of interest within which S has not read any object (i.e., O is outside the wall).

❷ *Brewer-Nash Write Rule*: subject S can write object O only if

- S can read O by Brewer-Nash read rule, and
- no object can be read which belongs to a different entity of the one for which write access is requested.

The *Brewer-Nash Read Rule* conveys the dynamic aspects of the Chinese Wall policy. The *Brewer-Nash Write Rule* is brought in to prevent subjects with Trojan horses from breaking the Chinese Walls.
In order to model the Chinese Wall policy by using temporal logic constraints we have to slightly modify some concepts. The only kind of access we consider are requests for services (i.e., method calls), thus we lose the distinction between write and read access, and the two Brewer-Nash rules are naturally merged. Let us consider the entities working in the system partitioned into two sets: $\mathsf{Clients}$, and $\mathsf{Servers}$, with the obvious meaning;

$$\mathsf{COI} = \{\mathsf{COI}_1, \ldots, \mathsf{COI}_n\}, \text{ and } \bigcup_{i=1}^{n} \mathsf{COI}_i = \mathsf{Servers}$$

where each COI_i represents a conflict of interest class. Thus the Brewer-Nash rules become:

- when $\mathsf{C}\in\mathsf{CLIENTS}$ asks for a service from $\mathsf{S}\in\mathsf{COI}_i$, that service is allowed if C already asked a service from S, or C never asked a service from an entity belonging to COI_i, but different from S.

This rule is expressed by the following constraint:

$$\exists \mathsf{m}_1.\Diamond_{\mathsf{p}} \langle \lambda x.x = \mathsf{C} \blacktriangleright \mathsf{S}[\,\mathsf{m}_1\,] \rangle \vee \big(\mathsf{S} \in \mathsf{COI}_i \wedge \big(\forall \mathsf{S}_1 \in \mathsf{COI}_i \wedge \mathsf{S}_1 \neq \mathsf{S}.\ \nexists \mathsf{m}_1.\big(\Box_{\mathsf{p}} \langle \lambda x.x = \mathsf{C} \blacktriangleright \mathsf{S}_1[\,\mathsf{m}_1\,] \rangle\big)\big)\big) \Rightarrow \mathsf{C} \blacktriangleright \mathsf{S}[\,\mathsf{m}\,]$$

This simple constraint realizes the Chinese Wall policy and it avoids information smuggling among entities belonging to the same conflict of interest class.

7 Related and Future Works

Escort and its *path-based* security architecture [16] have inspired our work. In Escort each module communicates with another module only through a special module, called a filter. Filters implement access control policies. A path is represented by the communications performed in order to complete the initial request. Each path can be interpreted as the history of a service execution. We abstracted filter and path mechanisms by mapping them, into channels and into history, respectively, in order to add flexibility (using reflection we can dynamically reconfigure the access control protocol, substituting any part: constraints set, policy realized, and so on), and to free the idea from the operating system context applying it to any system (separation of concerns, typical reflective feature, makes easier to separate the security mechanism from the rest of the application).

In [3] and in [14] reflective access control mechanisms have been proposed. In both cases they enforce role-based access control policies, realizing some advantages (listed in Section 4) due to the application of the reflective approach. Of this two papers, only [3] uses a communication-oriented reflective approach. The possibility of dynamically adapting and reconfiguring security modules through reflection has been analyzed in [19]. All the results determined in these works have been used in the current paper, adapting them to the context of history-dependent access control policies.

In the future, we intend to change the mechanism (see Section 5.2) used by each channel to retrieve and to rebuild the history before each constraint evaluation, from *on-demand* to *progressive*. For example, we have noted while preparing this paper, that it is possible to determine at compile-, or bootstrapping-time which channels need to know the history fragment built by each channel (it is enough to scan the complete constraints set, tracing all services which, to be allowed, need that history fragment), and using that information, each history change is propagated to those channels. In this way, we should achieve a performance improvement, because in the evaluation phase we avoid the overhead due to historical environment rebuilding, and history updating can be done during the channels' idle periods.

An open issue of this approach is represented by the system's history uncontrolled growth. In non-stopping or persistent systems, the history can always be useful, thus it cannot be deleted, and its dimension can become intractable. In order to avoid such a situation, it is necessary to find some deletion rules which permit to eliminate old, and hopefully, useless fragments of history.

We think that our realization of history-dependent access control through reflection is very flexible and adaptable to other history-dependent access control models. In order to prove this statement, in addition to modeling the Chinese Wall policy (see Sect. 6), we are planning to adapt it to history-dependent policies based on order logic instead of temporal logic (such as in [4]). This adaptation can be made simply by changing the constraints evaluation module in order to cover history intervals instead of history moments.

8 Conclusion

Using computational reflection we can separate the authorisation control mechanism from the application. This separation has the significant advantage of reducing the number of access-points of the security module to one (the shift-up action), making easier to protect it from malicious intrusions and attacks from the base-level or from other processes. A simple protection mechanism for the shift-up action could consist in executing the meta-level in a different address space as an independent process, communicating with the base-level via a mechanism similar to the *local procedure calls* (LPC) of Windows NT [8]. LPC is a locally optimized form of the well known mechanism of *remote procedure call* (RPC) of Unix and other systems: LPC is a message-passing mechanism through which clients make requests to servers and it is also used for the server's reply. The main necessary feature of the LPC mechanism is the protection domain installed around an LPC call.

Reflection offers several advantages when used to model authorisation mechanisms. Its main advantages are separation of concerns, transparency and modularity. Due to transparency, each communication is implicitly reified by a ValidationChannel, and it is validated before the related service is allowed. This behavior protects the system against external and internal attacks. Moreover, due to separation of concerns and modularity, the authorisation mechanisms can be designed within the application from early development stages, while maintaining them separate both from the logical and implementative point of view. This fact improves reusability of both functional and authorisation software and supports an independent testing of both.

Obviously, there are also drawbacks: one is a reduced execution efficiency: flexibility costs in efficiency. A second problem is represented by the protection mechanism around the authorisation layer (meta-level). Running it in a different address space may make programs too inefficient for most applications. Thus, more efficient protection mechanisms, not performing a complete context switching, should be designed. Hardware capability systems appear promising for this purpose.

Acknowledgments

Special thanks to Dr. Eva Coscia for her help with temporal logic and to Daniela Olivieri for the prototype implementation.

References

[1] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: A Reflective Model for Distributed Computation. In *proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, 98CH36191, pages 32–36, Phoenix, Arizona, USA, on 16th-18th February 1998. IEEE.

[2] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective Authorization Systems. In *proceedings of ECOOP Workshop on Distributed Object Security (EWDOS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), pages 35–39, Brussels, Belgium, on 20th-24th July 1998. Unité de Recherche INRIA Rhŏne-Alpes.

[3] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective Authorization Systems: Possibilities, Benefits and Drawbacks. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science, pages 35–51. Springer-Verlag, July 1999.

[4] Elisa Bertino, Francesco Buccafurri, Elena Ferrari, and Pasquale Rullo. An Authorization Model and Its Formal Semantics. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Lecture Notes in Computer Science*, number 1185 in Proceedings of 5th European Symposium on Research in Computer Security (ESORICS'98), pages 127–142, Louvain-la-Neuve, Belgium, September 1998. Springer-Verlag.

[5] Forrest D. Brewer and Jonathan M. Nash. The Chinese Wall Security Policy. *IEEE Symposium on Security and Privacy*, pages 215–228, 1989.

[6] Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.

[7] Gerardo Costa and Gianna Reggio. Specification of Abstract Dynamic DataTypes: A Temporal Logic Approach. *Teoretical Computer Science*, 173(2), 1997.

[8] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[9] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-Based Access Control for Mobile Code. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science, pages 415–434. Springer-Verlag, July 1999.

[10] Eduardo B. Fernandez, Rita C. Summers, and Christopher Wood. *Database Security and Integrity*. Addison-Wesley, Reading, Massachusetts, 1981.

[11] Rex H. Hartson and David K. Hsiao. Full Protection Specification in the Semantic Model for Data Base Protection Languages. *Proceedings of ACM*, pages 90–95, 1976.

[12] Butler W. Lampson. Protection. *Operating System Review*, 8(1):18–34, January 1974. Reprint.

[13] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*, pages 147–156. ACM, October 1987.

[14] Thomas Riechmann and Jürgen Kleinöder. Meta-Objects for Access Control: Role-Based Principals. In Colin Boyd and Ed Dawson, editors, *Lecture Notes in Computer Science*, number 1438 in Proceedings of 3rd Australasian Conference on Information Security and Privacy (ACISP'98), pages 296–307, Brisbane, Australia, July 1998. Springer-Verlag.

[15] Ravi Sandhu. Lattice-Based Enforcement of Chinese Walls. *Computers and Security*, 11(8):753–763, December 1992.

[16] Oliver Spatscheck and Larry L. Peterson. Escort: A Path-Based OS Security Architecture. Technical Report TR-97-17, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, November 1997.

[17] Colin Stirling. Modal and Temporal Logic. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook in Computer Science*, volume 2, pages 477–563. Claredon Press, Oxford, 1992.

[18] Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.

[19] Robert J. Stroud and Ian Welch. Dynamic Adaptation of the Security Properties of Application and Components. In *proceedings of ECOOP Workshop on Distributed Object Security (EWDOS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), pages 41–46, Brussels, Belgium, July 1998. Unité de Recherche INRIA Rhǒne-Alpes.

A Constraints Semantics

The historical evolution of the system authorisation can be modeled by a *dynamic algebra* $\mathcal{H}_{\mathsf{DA}}$ on the *dynamic signature*[2]

$$\mathcal{H}_{\mathsf{D}\Sigma} = (\,\mathsf{Sorts}, \mathsf{Operations}, \mathsf{Predicates}\,)$$

where

- $\mathsf{Objects}$, $\{\mathsf{Methods}\}$, $\mathsf{Args} \subseteq \mathsf{Sorts}$; $\mathsf{Objects}$, and Args are classes of sorts, respectively, for the entities involved in the requests and for the arguments passed in the requests; $\mathsf{Methods}$ describes the services offered;
- $\mathsf{History} \in \mathsf{Sorts}$ is a dynamic sort, its elements are denoted by $\sigma_1, \sigma_2, \sigma_3, \ldots$, and $lab(\,\mathsf{History}\,) \equiv \Delta$;
- In $\mathsf{Operations}$ there are only zerary operators representing the elements of the sorts;
- $\{\,\_ \blacktriangleright \_[\,\_(\,\_\,)\,] : \mathsf{O}_1 \times \mathsf{O}_2 \times \mathsf{Methods} \times \mathsf{A}\,\} \subseteq \mathsf{Predicates}$, for all $\mathsf{O}_1, \mathsf{O}_2 \in \mathsf{Objects}$ and for all $\mathsf{A} \in \mathsf{Args}$

then, the triple $\mathcal{H}_{\mathcal{LTS}} = (\,\mathsf{History}, \Delta, \rightarrow)$ is a *labeled transition system*[3].
The different possible evolutions of the dynamic elements are represented by the maximal sequences of states and labels of the form

$$\cdots \sigma_{-2} \xrightarrow{\delta_{-2}}_{\mathcal{H}_{\mathsf{DA}}} \sigma_{-1} \xrightarrow{\delta_{-1}}_{\mathcal{H}_{\mathsf{DA}}} \sigma_0$$

$\mathsf{PATH}\,(\,\mathcal{H}_{\mathsf{DA}}, \mathsf{History}\,)$ is the set of all sequences having either of the two forms below:

- $\cdots \sigma_{-4}\delta_{-4}\sigma_{-3}\delta_{-3}\sigma_{-2}\delta_{-2}\sigma_{-1}\delta_{-1}\sigma_0$ (infinite path)
- $\sigma_{-k}\delta_{-k}\sigma_{-(k-1)}\delta_{-(k-1)}\sigma_{-(k-2)}\delta_{-(k-2)} \cdots \sigma_{-1}\delta_{-1}\sigma_0\; k \geq 0$ (finite path)

where for all $i \in \mathbb{Z}^-\, \sigma_i \in (\,\mathcal{H}_{\mathsf{DA}}\,)_{\mathsf{History}}$, $\delta_i \in (\,\mathcal{H}_{\mathsf{DA}}\,)_{lab(\mathsf{History})}$ and $(\,\sigma_i, \delta_i, \sigma_{i+1}\,) \in \rightarrow_{\mathcal{H}_{\mathsf{DA}}}$, and in the case of finite path there are no σ, δ such that $(\,\sigma, \delta, \sigma_{-k}\,) \in \rightarrow_{\mathcal{H}_{\mathsf{DA}}}$. For each path ρ

- $\sigma(\,\rho\,)$ denotes the last state of ρ : σ_0
- $\delta(\,\rho\,)$ denotes the last label of ρ : δ_{-1}, if it exists
- $\rho_{|n}$ denotes the path $\cdots \sigma_{n-3}\delta_{n-3}\sigma_{n-2}\delta_{n-2}\sigma_{n-1}\delta_{n-1}\sigma_n$ if it exists

In our concern, the relevant information of the LTS are the labels. Each path $\rho_{|lab(\mathsf{History})}$, obtained from ρ considering only the labels, is equivalent to a system history $\hbar$.
Each constraint is expressed by a formula on the system evolution such as

$$\Gamma \Rightarrow \delta$$

where Γ is a first-order branching temporal logic [17] formula, which evaluation can involve the history of system interactions and δ is a request which authorisation depends on the evaluation of the constraint Γ.

[2] In accordance with [7] a *dynamic signature* $\mathsf{D}\Sigma$ is a pair $(\,\Sigma, \mathsf{DS}\,)$ where:

- $\Sigma = (\,\mathsf{Sorts}, \mathsf{Operations}, \mathsf{Predicates}\,)$ is a predicate signature,
- $\mathsf{DS} \subseteq \mathsf{Sorts}$ (the elements in DS are the dynamic sorts, ie., the sorts of dynamic elements),
- for all $\mathsf{ds} \in \mathsf{DS}$ there exists a sort $lab(\,\mathsf{ds}\,) \in \mathsf{Sorts} \setminus \mathsf{DS}$ and a predicate symbols $\rightarrow : \mathsf{ds} \times lab(\,\mathsf{ds}\,) \times \mathsf{ds} \in \mathsf{Predicates}$.

A *dynamic algebra* on $\mathsf{D}\Sigma$ is just a Σ-algebra; the term algebra $T_{\mathsf{D}\Sigma}(\,X\,)$ is just $T_\Sigma(\,X\,)$

[3] A *labeled transition system* is a triple

$$\mathcal{LTS} \equiv (\,\mathsf{States}, \mathsf{Labels}, \rightarrow)$$

where States is the set of intermediate states on which the system can evolve, Labels is the set of labels describing the possible state transitions, and $\rightarrow : \mathsf{States} \times \mathsf{Labels} \rightarrow \mathsf{States}$ is a function describing the system evolution.

Due to the nature of the constraints (i.e., related to events yet occurred) we have to express, they can be expressed by using only connectives in the past; thus the formulas' syntax will be expressed by:

$$\mathsf{DF} \;::=\; \mathrm{P} \,\big|\, \Delta \,\big|\, \neg\mathsf{DF} \,\big|\, \mathsf{DF}_1 \wedge \mathsf{DF}_2 \,\big|\, \mathsf{DF}_1 \vee \mathsf{DF}_2 \,\big|\, \forall x.\mathsf{DF}(x) \,\big|\, \exists x.\mathsf{DF}(x) \,\big|\, \triangle(t, \mathsf{PF}) \,\big|\, \triangledown(t, \mathsf{PF})$$

$$\mathsf{PF} \;::=\; \neg\mathsf{PF} \,\big|\, \mathsf{PF}_1 \wedge \mathsf{PF}_2 \,\big|\, \mathsf{PF}_1 \vee \mathsf{PF}_2 \,\big|\, \forall x.\mathsf{PF}(x) \,\big|\, \exists x.\mathsf{PF}(x) \,\big|\, [\lambda x.\mathsf{DF}(x)] \,\big|\, \langle \lambda x.\mathsf{DF}(x) \rangle \,\big|\, \circ_{\mathrm{p}} \mathsf{PF} \,\big|\, \diamond_{\mathrm{p}} \mathsf{PF} \,\big|\, \square_{\mathrm{p}} \mathsf{PF} \,\big|\, \mathsf{PF}_1 \,\mathcal{S}\, \mathsf{PF}_2$$

where DF, PF describe, respectively, dynamic, and path formulas, P represents boolean predicates, Δ represents the requests, and t is a term belonging to $\mathcal{H}_{\mathsf{D\Sigma}}$.
We adopt the following interpretation for dynamic (φ_i) and path formulas (π_i) in the $\mathsf{D\Sigma}$-algebra $\mathcal{H}_{\mathsf{DA}}$. $V : X \to \mathcal{H}_{\mathsf{DA}}$ is a variable evaluation.

dynamic formulas

$\mathcal{H}_{\mathsf{DA}}, V \models P(t_1, \ldots, t_n)$	iff	$\left(t_1^{\mathcal{H}_{\mathsf{DA}},V}, \ldots, t_n^{\mathcal{H}_{\mathsf{DA}},V}\right) \in P^{\mathcal{H}_{\mathsf{DA}}}$
$\mathcal{H}_{\mathsf{DA}}, V \models \neg\varphi$	iff	$\mathcal{H}_{\mathsf{DA}}, V \not\models \varphi$
$\mathcal{H}_{\mathsf{DA}}, V \models \varphi_1 \wedge \varphi_2$	iff	$\mathcal{H}_{\mathsf{DA}}, V \models \varphi_1$ and $\mathcal{H}_{\mathsf{DA}}, V \models \varphi_2$
$\mathcal{H}_{\mathsf{DA}}, V \models \forall x.\varphi$	iff	for all $v \in \mathcal{H}_{\mathsf{DA}\,srt}$, with *srt* sort of x, $\mathcal{H}_{\mathsf{DA}}, V[v/x] \models \varphi$
$\mathcal{H}_{\mathsf{DA}}, V \models \triangle(t, \pi)$	iff	$t^{\mathcal{H}_{\mathsf{DA}},V}$ is defined and for all $\rho \in \mathsf{PATH}(\mathcal{H}_{\mathsf{DA}}, \mathsf{History})$, with $\mathsf{History}$ sort of t, if $\sigma(\rho) = t^{\mathcal{H}_{\mathsf{DA}},V}$ then $\mathcal{H}_{\mathsf{DA}}, \rho, V \models \pi$

path formulas

$\mathcal{H}_{\mathsf{DA}}, \rho, V \models [\lambda x.\varphi]$	iff	$\mathcal{H}_{\mathsf{DA}}, V[\sigma(\rho)/x] \models \varphi$
$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \langle \lambda x.\varphi \rangle$	iff	either $\mathcal{H}_{\mathsf{DA}}, V[\delta(\rho)/x] \models \varphi$ or $\delta(\rho)$ is not defined
$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \neg\pi$	iff	$\mathcal{H}_{\mathsf{DA}}, \rho, V \not\models \pi$
$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \pi_1 \wedge \pi_2$	iff	$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \pi_1$ and $\mathcal{H}_{\mathsf{DA}}, \rho, V \models \pi_2$
$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \forall x.\pi$	iff	for all $v \in \mathcal{H}_{\mathsf{DA}\,srt}$, with *srt* sort of x, $\mathcal{H}_{\mathsf{DA}}, \rho, V[v/x] \models \pi$
$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \diamond_{\mathrm{p}} \pi$	iff	exists j such that $\rho_{\mid j}$ is defined, $\mathcal{H}_{\mathsf{DA}}, \rho_{\mid j}, V \models \pi$
$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \circ_{\mathrm{p}} \pi$	iff	either $\rho_{\mid -1}$ is not defined, or $\mathcal{H}_{\mathsf{DA}}, \rho_{\mid -1}, V \models \pi$
$\mathcal{H}_{\mathsf{DA}}, \rho, V \models \pi_1 \,\mathcal{S}\, \pi_2$	iff	exists j such that $\rho_{\mid j}$ is defined, $\mathcal{H}_{\mathsf{DA}}, \rho_{\mid j}, V \models \pi_2$ and for all h such that $j < h$, and $\rho_{\mid h}$ is defined $\mathcal{H}_{\mathsf{DA}}, \rho_{\mid h}, V \models \pi_1$

The remaining connectives are defined by $\varphi_1 \vee \varphi_2 \overset{\text{def.}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\exists x.\varphi \overset{\text{def.}}{=} \neg\forall x.\neg\varphi$, $\triangledown(t, \pi) \overset{\text{def.}}{=} \neg\triangle(t, \neg\pi)$, and $\square_{\mathrm{p}} \pi \overset{\text{def.}}{=} \neg\diamond_{\mathrm{p}} \neg\pi$.
Requests' validity implies determining predicates' validity and corresponds to determining their belonging to the history of the system.