Formal Specification of an Access Control System

Susan Stepney and Stephen P. Lord

GEC-Marconi Research Centre, Chelmsford,UK.

SUMMARY

Computing facilities networked together but controlled by different administrations pose a problem of access control. Who decides who can use what?

We specify a formal model for an access control system which allows users and services from different administrations to communicate with each other, while still allowing the administrators to retain control of their own parts of the network. The model, written in the Z specification language, has been developed as the access control system for ADMIRAL, though it is not specific to ADMIRAL. It provides a framework for administrators to build access control systems to meet their differing requirements.

A system based on the model would allow users to log in to a distributed computing system and to make requests for services in any part of the system, without having to provide any more information about themselves. After this initial log in all subsequent access control decisions are handled automatically, and remain invisible to the user unless access is refused.

We also discuss the experience we have had animating this model in Prolog.

KEY WORDS Networks Access control Formal model Z specification Prolog

1. Objectives of the Model

In a conventionally organized network, very little attention is paid to access control on a network-wide basis. Usually individual machines provide their own facilities. This can lead to frustration both for users and for administrators; users are continually faced with log-in prompts, and administrators have to control a multitude of tables on different machines.

Most access control schemes designed for networks are based around a centralized service, or are controlled by a single administration. These fail to address the problems associated with access control in a multi-administration network. In such a network, several autonomous access control system (ACSs) have to interact.

The model described here has been developed as the access control system for project ADMIRAL. ADMIRAL is a collaborative project supported by the Alvey programme, carrying out research into the use and management of high performance networks. An internet of linked networks, covering five industrial and academic sites, is being set up. The project is described in Reference 1.

Our model is not specific to ADMIRAL, however. A system based on our model will have the following properties:

1. Autonomous administrations can work with each other, but still retain control over their own facilities.
2. Users' access to services can be controlled, even when the user and the service fall under different administrations. This control is invisible to users, unless they try to access services not available to them.
3. An administrator can make use of another's facilities, if they both agree.
4. Multiple levels of security are available to users and administrations. They can insist on a particular level for certain operations.

Susan Stepney and Stephen P. Lord. "A Formal Specification of an Access Control System." Software—Practice and Experience, Vol 17(9), p575–593, 1987

We do not address the problem of authentication in this paper. However, the model does highlight the points where authentication is necessary.

In the next section we give a brief informal overview of the model, to provide a framework for the formal description in later sections. After the formalism, we describe our experiences animating the model in Prolog.

2. Overview of the Model

In this section we give a brief, informal overview of the model before plunging into the formalism of the next sections. A more detailed informal description of the model is given in Reference 2.

Principals make requests for services. Servers provide services. A Client handles a request for a Principal, and passes it on to a Server. Principals are responsible for requests; Clients and Servers are the communicating parties involved in requests. Principals have access rights, permissions to make certain requests of certain Servers.

Authorities provide the access control functions. They store Statements about Principals' access rights. Statements say things like 'JOHN has READ access to FILEx', 'JANE has PRINT access to PRINTERa'.

Authorities are used by Servers to check the access rights of Principals, and by Clients to gain access to Servers for Principals. Authorities can obtain and generate Statements for other Authorities and for Entities. Each Client and Server has a Local Authority, which it trusts to make appropriate access control decisions. These Entities will not use any other Authority directly.

When a Client makes a request to a Server, Authorities check the Principal's right to the Server. These checks are based on Statements about the Principal, Server and request.

Trust allows one Authority to use Statements made by another Authority. In order for A to accept a Statement about a Server from B, A must trust B to provide correct Statements about that Server. Since Statements can be passed between Authorities, it is possible for Principals controlled by one Authority to have access to Servers controlled by other Authorities, provided that they have the appropriate permission.

The request to a Server, together with the subsequent actions, is known as a Transaction. A Transaction starts when a request is made, and finishes when the request has been dealt with, or refused. It includes access control decisions.

Figure 1 shows the communication paths between the Entities and the Authorities involved in the access control. Some of these paths are used when collecting Statements, others are used for the Transactions.

The Principal is responsible for the request (1) ; the Client makes the request on the Principal's behalf (2). The request goes via the Client's Local Authority (CLA). The CLA holds cached statements about the Principal and Server. These are obtained both from its own store and, optionally, from other trusted Authorities (3).

The request is passed on (4) to the Server's Local Authority (SLA). The SLA checks the access right, using its cached Statements. These are obtained from the CLA's cache, and, optionally, from its own store and from other Trusted Authorities (5). If the access conditions have been fulfilled, the request is passed on to the Server for processing (6). Further exchange of data may occur (7).

In the model the gathering of Statements and the making of a request are treated as logically separate issues. In practice, the first request to a Server could trigger the other activities, and they could occur in parallel.

Different qualities of service (qos) are offered by the model. The qos can be related to the level of security provided by the underlying mechanisms.

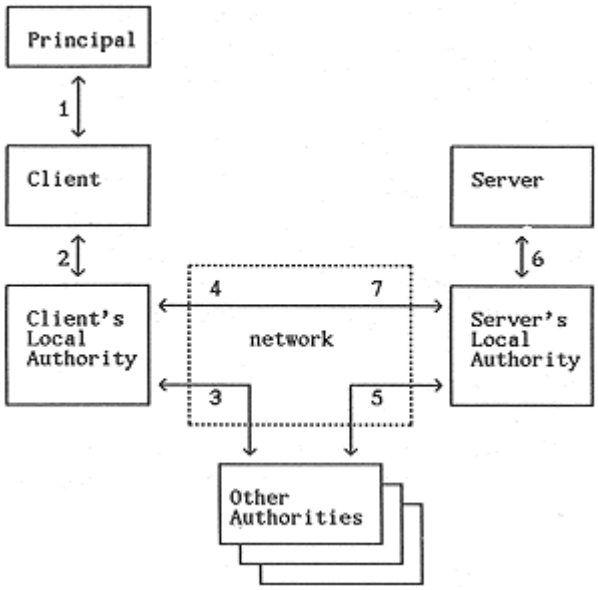


Figure 1. The communication paths in a Transaction

3. Formal Model and Z

In order to provide a complete and consistent definition of a computer security and access control system, there is a need to develop a formal model. Formalism also allows proofs of desired properties of the system. Two papers that describe and review attempts to formalize security systems are References 3 and 4. A more rigorous definition can be provided by using a formal language.

We give a formal specification of an access control system,[1] using Z. This is a general purpose specification language, using mathematics to specify the system, and schemas to structure and modularize the specification.

A Z specification consists of a mix of formal mathematics and informal English text. The formal part provides the precise, unambiguous definition of the system. The informal text acts as a commentary on the formal part. It can be consulted to find out what aspects of the real world are being described.

A Z specification describes the state of the system, and various operations that can change the state. It uses schema boxes to display this information. A schema box has two parts:

```
Name
signature
---------
predicates
```

The signature gives the types of the variables used in the schema. The predicates give the conditions these variables have to satisfy for the schema to be true. The 'before' state of the schema is indicated by unprimed variables, the 'after' by primed variables.

[1] The full model includes other components not described here. These are auditing system for recording access control decisions, and an association system which ensures that communicating parties have authenticated themselves to each other.

Such a short description cannot do justice to Z, but a specification is fairly readable, even without an understanding of the details. A Z specification of a well-known system is described in Reference 5. The Z language is described in Reference 6, and the notation used in this paper is summarized in the Appendix.

4. Entities

An Entity is something involved in the provision or use of a service. It could be a person or a software module. Entities are the objects controlled and protected by the ACS.

```
[ENTY]
```

ENTY is the set of all Entities known to the system.

4.1 Principals, Clients and Servers

An Entity which initiates a request for a service, either explicitly or implicitly due to a previous request it made, is called a Principal. When an Entity responds to a request it is called a Server. A request is handled by an Entity called a Client, and passed on to a Server. Principals are responsible for requests; Clients and Servers are the communicating parties involved in requests. Access rights depend on attributes of Principals, not of Clients.

Entities are not fixed in the role of Principal, Client or Server. For example, a Server might receive a request from a Principal which requires it to make a request of another Server. It could then become a Principal in its own right, or a Client for the original Principal, in order to make the subsequent request. A software Entity could well be Principal and Client simultaneously. Similarly, the Client could also be a Server responding to another request.

Software might take on any of the roles. People will tend to be Principals.

5. Requests

Entities make requests for services.

```
[RQST]
```

RQST is the set of all requests that can be made.

Different Servers will respond to different requests (see section 10). For example, a file-server might respond to read, write, delete; a time-server to now, reset. Principals have access rights, permissions to make certain requests of Servers (see section 8). For example, a particular principal could have the right to consult a time-server for the current time, but not to reset it.

6. Authorities

Authorities provide the access control functions.

```
[AUTY]
```

AUTY is the set of all Authorities known to the system.

Authorities store Statements about Entities' access rights (see section 10), and can cache Statements obtained from other Authorities (see section 13.2) for consultation during a session of requests. They check that Principals fulfil servers' access conditions (see section 14.2).

Each Authority is controlled by a single administration; an administration may control more than one Authority. The administration says how its Authority will behave in response to requests from Clients, Servers, other Authorities and external administrations. This behaviour forms part of the administration's security policy.

When a Client makes a request to a Server, Authorities check the principal's right to the Server. These checks are based on Statements about the Principal, Server and request.

7. Quality of Service

The quality of service (qos) describes the level of security provided; for example, none, confidential, secret. (No particular meaning is attached to these levels in this paper.)

```
[QOS]
```

`QOS` is the set of all qualities of service. It is a set that can be ordered by the ≤ relation

```
∀ q1,q2,q3 : QOS •
  (q1 ≤ q2 ∨ q2 ≤ q1)
  ∧ (q1 ≤ q2 ∧ q2 ≤ q1 ⇒ q1 = q2)
  ∧ (q1 ≤ q2 ∧ q2 ≤ q3 ⇒ q1 ≤ q3)
```

7.1 Where quality of service occurs

Each Session has a qos, specified by the Principal (see sections 11 and 13.1). When the Principal logs on, authentication might be different for high qos's than for lower ones. Subsequent Transactions will be limited by the log-on qos.

Each Transaction has a qos, specified by the Principal or Client (see section 14.2). The Principal might be logged in at a high qos, but could specify that certain requests need only be fulfilled at a lower qos. The qos of a Transaction cannot be higher than that of the Session.

Each request has a qos, specified by the Server. Servers can insist that they will only respond to certain requests at a specified qos. For example, a time-server might be happy to supply the current time at any qos, but might insist that it can only be reset at the highest qos.

The qos required will depend on the nature of the request, and will be agreed by the Entities and Authorities involved. Usually only the Authorities will be involved in this (mandatory qos), but the Principal (see section 14.2) and Server (see section 10) are given the option of asking for a specific qos if there is something special about the request (discretionary qos). Not all protection mechanisms may be available everywhere in the DCS, and thus the requested qos may not be available. In these cases the request will fail; it can always be retried with a lower qos.[2]

[2] The authorities involved in a request are responsible for establishing and maintaining the required qos. If different authorities have different security policies, some mechanism might be necessary to agree on a common qos.

8. Statements

An Authority stores Statements about Principals' access rights to Servers. A Statement is:

```
STMT
issuing_auty : AUTY
principal : ENTY
server : ENTY
requests : ℙ RQST
```

A Statement consists of the issuing Authority (the Authority that stores it), a Principal (the Entity making a request), a Server (the Entity responding to a request), and those requests the Principal has permission to make of the Server.

Authorities check access rights by consulting their own Statements and those of other Authorities (see sections 13.2 and 14.2).

9. Trust

Trust allows one Authority to use Statements made by another Authority. In order for A to accept a Statement about a Server from B, A must trust B to provide correct Statements about that Server.

Trust need not be complete; an Authority can trust another to supply Statements about some Servers, but not about others. This enables Authorities to keep tighter control over particular resources. Trust need not be symmetric; A trusts B does not necessarily mean that B trusts A.

Since statements can be passed between Authorities, it is possible for Principals controlled by one Authority to have access to Servers controlled by other Authorities, provided they have the appropriate permission.

The setting up and changing of the trust relation is outside the model. It will be decided by (human) administrators, and is part of the security policy.

10. Access Control System

Now that we have some building blocks in the form of the sets ENTY, RQST, AUTY and QOS, we can start to define the state of the system. Using the concepts of Entities, Requests, Authorities, Statements and Trust introduced above, the access control system is:

```
ACS
auty_stmt : AUTY ⟷ STMT
enty_rqst : QOS ⟶ (ENTY ⟷ RQST)
trusted_by : AUTY ⟶ (AUTY ⟷ ENTY)
local_auty : ENTY ⤔ AUTY
---
∀ a : AUTY; s : STMT; rqst : RQST |
    (a,s) ∈ auty_stmt ∧ rqst ∈ s.requests •
  ∃ q : QOS • (s.server,rqst) ∈ enty_rqst q

∀ a : AUTY; s : STMT | (a,s) ∈ auty_stmt •
  s.issuing_auty = a

∀ a : AUTY; s : ENTY • (a,s) ∈ trusted_by a
```

This schema defines some relationships between the sets introduced earlier:

(a) `auty_stmt` relates an Authority and the Statements it holds in store. There can be Entities known to the system that are not mentioned in any Statement, for example Entities that are only ever Clients.

(b) `enty_rqst` relates a Server and the requests it will respond to at different qos's. A Server might respond to a request at a range of qos's. Entities that are never Servers (for example, people) will not appear in this relation.

(c) `trusted_by` relates an Authority to those Authorities it trusts to make Statements about particular Servers.

(d) `local_auty` relates Clients and Servers to their Local Authorities, which make the access control decisions. These Entities will not use any other Authority directly. This function is partial since Entities that are only ever Principals do not have a Local Authority.

The relationships are constrained by the predicates as follows:[3]

(1) The requests mentioned in Statements are possible requests for the relevant Server. No Statement will say, for example, that a Principal has the right to take the square root of a print-server, or append to a random number server!

(2) The issuing Authority of a Statement is the Authority where the Statement is stored. This can be thought of as the Authority's 'signature' on the Statement.

(3) All Authorities trust themselves to make Statements about any Server.

Note that in all the following schemas, the state of the ACS never changes. Changes to the trust relation, the Statements store, and so on, are outside the model.

11. Sessions

An Entity's session is the region of the DCS from which requests can be made with that Entity acting as Principal. A Session comes into existence when the Principal is first authenticated, and continues until the Principal can no longer use the DCS (for example, log-out for a person, termination for a software module). The region contained by the Session can vary. It expands to include those Entities whose Local Authorities have been satisfied with the Principal's access rights. It contracts as cached Statements are discarded.

Before a Server will respond to a request, it must be a member of the Principal's Session. It can become a member if its Local Authority can be satisfied that the Principal has access right to the Server (see section 13.3).

```
[SID]
```

`SID` is the set of all session ids. There is a special session id, the null id, which is never the id of any existing session. This can be used to record an unsuccessful log-in attempt:

```
| NullSid : SID
```

The Session system is

[3] Note: in Z the application of function `m` to argument `x` can be written either as `m x` or as `m(x)`. The former is preferred; the latter can be used to improve clarity in complicated expressions.

```
SESS
owner : SID ⇸ ENTY
members : SID ⇸ ℙ ENTY
cache : SID ⇸ (AUTY ⟷ STMT)
qos : SID ⇸ QOS
---
dom qos = dom members = dom owner = dom cache
```

owner relates a session id to the Principal which owns the Session: members relates a session id and those Clients and Servers whose Local Authorities have been satisfied that the owner has the relevant access permissions.

cache relates a session id, an Authority and the Statements it has cached about the Session. The cached Statements are used for determining the access rights of the Session's Principal. Different Authorities will normally have different caches of Statements referring to the same Session. Note that, strictly, caches are an efficiency consideration, but they have been included at this top level since they are considered to be an important feature of the design.

qos relates a Session id to the Session's quality of service.

All existing sessions have an owner, a set of members, a cache (possibly empty) and a quality of service.

12. The Distributed Computing System

The state of the Distributed Computing System is the combination of the (unchanging) Access Control System and the (changeable) Session System

```
DCS
ACS
SESS
---
ran (⋃ (ran cache)) ⊆ ran auty_stmt
```

The DCS automatically inherits the predicates from its subsystems. The extra predicate ensures that all cached Statements are also stored by some Authority.

12.1 Other notation

```
ΞDCS ≙ [ DCS; DCS′ | θ DCS′ = θ DCS ]
```

ΞDCS describes a before and after[4] DCS that is unchanged.

```
ΔDCS ≙ [ DCS; DCS′ | θ ACS′ = θ ACS ]
```

ΔDCS describes a change in the Session system, but not the Access Control system. In most cases where the Session system is changed (log-in is the exception) the change refers to an existing Session and particular Principal. OSESS includes the predicates that ensure that the relevant Session exists, and is owned by the Principal.[5]

[4] Recall that unprimed variables refer to *before* states, and primed variable to *after* states.

[5] It is conventional in Z to end input variable names with a ?, and end output variable names with a !.

```
┌─ΔSESS──────────────────────────────────────────────┐
│ ΔDCS                                               │
│ sid? : SID                                         │
│ p? : ENTY                                          │
├────────────────────────                            │
│ sid? ∈ dom owner                                   │
│ p? = owner sid?                                    │
└────────────────────────────────────────────────────┘
```

In the above cases, the predicates on the DCS are inherited automatically.

13. Changing Sessions

13.1 Starting a Session-Log-in

Successful-login (Figure 8) sets up a new session for a principal `p?` [6] at qos `q?`. It allocates a new session id, `sid!`. There is some log-in authentication server, `s?`, that can be used by Principals, known or unknown. Only the Session system is changed.

```
┌─Successful_login───────────────────────────────────┐
│ ΔDCS                                               │
│ p? : ENTY ∪ UNKNOWN_ENTY                           │
│ s? : ENTY                                          │
│ q? : QOS                                           │
│ sid! : SID                                         │
├────────────────────────                            │
│ p? ∈ ENTY                                          │
│                                                    │
│ sid! ≠ NullSid                                     │
│ sid! ∉ dom owner                                   │
│                                                    │
│ owner′ = owner ∪ {sid! ↦ p?}                       │
│ members′ = members ∪ {sid! ↦ {s?}}                 │
│ cache′ = cache ∪ {sid! ↦ {}}                       │
│ qos′ = qos ∪ {sid! ↦ q?}                           │
└────────────────────────────────────────────────────┘
```

(1) The Principal must be known to the system.
(2) The new session id is neither the `NullSid`, nor one that has already been used.
(3) A new Session is created with `p?` as the owner, `s?` as the only member, no cached Statements, and quality of service `q?`.

Unsuccessful-login describes the case where the Principal is not known to the system. The state of the DCS is unchanged. The Session id is the special `NullSid`.

[6] In this schema, and most of the following ones, it is required that the Principal be known to the system, that is `p? ∈ ENTY`. This will initially be checked by something like name and password. Subsequently, the log-in authentication can be trusted by other Authorities, or they might do their own authentication.

```
Unsuccessful_login
ΞDCS
p? : ALL_ENTY
q? : QOS
sid! : SID
─────────────────
p? ∈ UNKNOWN_ENTY
sid! = NullSid
```

The complete Login definition is a combination of the above schemas:

```
Login ≙ Successful_login ∨ Unsuccessful_login
```

Login is either a successful log-in, or an unsuccessful log-in. Unsuccessful attempts are distinguished by the `NullSid`.

13.2 Gathering Statements

An Authority can get a Statement from its own store, another Authority's store, or another Authority's cache.

The next schema describes adding a Statement to a cache. The two following schemas will specify which Statement is to be cached.

`Add_Statement_to_Cache` changes the Session system by adding a Statement to the cache for Session `sid?`. The Statement is added to Authority `a?`'s part of the cache, and was gathered from `b?`'s store.

```
Add_Statement_to_Cache
ΔSESS
s? : ENTY
a?,b? : AUTY
stmt : STMT
─────────────────
(b?,s?) ∈ trusted_by a?

cache'
 = cache ⊕ {sid? ↦ cache sid? ∪ {(a?,stmt)}}

owner' = owner
members' = members
qos' = qos
```

(1) The predicate ensures that `a?` trusts `b?` to make Statements about `s?`. Note that the Statements are gathered at the Session's qos, not at the qos of the request (see section 14.2) .
(2) The Statement is added to the consulting Authority's cache.
(3) All other parts of the system are unchanged.

The case `a? = b?` describes an Authority consulting its own stored Statements. This is always possible, since all Authorities trust themselves to make Statements.

There are two possible places this Statement could come from. One is an Authority's store of Statements: `Get_Statement_from_Store` gets a Statement about a Principal `p?` and a Server `s?` from `b?`'s store.

```
┌─Get_Statement_from_Store──────────────────────────┐
│ DCS                                               │
│ p?,s? : ENTY                                      │
│ b? : AUTY                                         │
│ stmt : STMT                                       │
├───────────────────────                            │
│ (b?,stmt) ∈ auty_stmt                             │
│ stmt.principal = p?                               │
│ stmt.server = s?                                  │
└───────────────────────────────────────────────────┘
```

The other place the Statement could come from is a cache: `Get_Statement_from_Cache` gets a Statement about a Principal `p?` and a Server `s?` from `b?`'s cache relating to Session `sid?`.

```
┌─Get_Statement_from_Cache──────────────────────────┐
│ DCS                                               │
│ sid? : SID                                        │
│ p?,c?,s? : ENTY                                   │
│ a?,b? : AUTY                                      │
│ stmt : STMT                                       │
├───────────────────────                            │
│ a? = local_auty s?                                │
│ b? = local_auty c?                                │
│                                                   │
│ (b?,stmt) ∈ cache sid?                            │
│ stmt.principal = p?                               │
│ stmt.server = s?                                  │
│                                                   │
│ (stmt.issuing_auty,s?) ∈ trusted_by a?            │
└───────────────────────────────────────────────────┘
```

(1) The only time an Authority `a?` can get a Statement from another Authority `b?`'s cache is if `a?` is the Server's Local Authority (SLA) and `b?` is the Client's Local Authority (CLA).
(2) The CLA has a Statement in its cache for this Session that is about the Session's owner and the relevant Server.
(3) The SLA trusts the Authority that made this Statement. Note that when this schema is combined with `Add_Statement_to_Cache`, there will also be the condition that the SLA trusts the CLA to make Statements about the Server.

The complete specification for getting a Statement is[7]

```
Get_Statement ≙ ((Get_Statement_from_Cache
                  ∨ Get_Statement_from_Store)
                 ∧ Add_Statement_to_Cache) \ (stmt)
```

13.3 Expanding a Session

`Expand_Session` changes the Session system by adding Server `s?` as a member of the Principal's Session. `a?` is the Authority that makes the decision to expand the Session.

[7] The expression `\ (stmt)` 'hides' the variable `stmt` (see Appendix).

```
┌─Expand_Session──────────────────────────────────────────┐
│ ΔSESS                                                   │
│ s? : ENTY                                               │
│ a? : AUTY                                               │
├─────────────────────                                    │
│ s? ∉ members sid?                                       │
│ a? = local_auty s?                                      │
│                                                         │
│ ∃ stmt : STMT | (a?,stmt) ∈ cache sid? •                │
│   stmt.principal = p? ∧ stmt.server = s?                │
│ members'                                                │
│  = members ⊕ {sid? ↦ members sid? ∪ {s?}}                │
│                                                         │
│ owner' = owner                                          │
│ cache' = cache                                          │
│ qos' = qos                                              │
└─────────────────────────────────────────────────────────┘
```

(1) The Server is not already in the Session, and the Authority must be the Server's Local Authority.
(2) If there is a Statement in the Authority's cache that refers to the Principal and Server, then the Server is added to the Session's members.
(3) The rest of the system is unaltered.

Note that the SLA is guaranteed to trust the Statement, since this was checked before the Statement was allowed into the SLA's cache. The only Statements in an Authority's cache are ones made by trusted Authorities:

```
DCS; sid : SID; a : AUTY; stmt : STMT |
  (a,stmt) ∈ cache sid
⊢
(stmt.issuing_auty,stmt.server) ∈ trusted_by a
```

13.4 Shrinking a Session—discarding a cached Statement

Cached statements might be discarded because of time-out or space constraints. The Session is modified suitably.

The cached Statement to be discarded is `stmt?`, part of `sid?`'s cache at Authority `a?`.

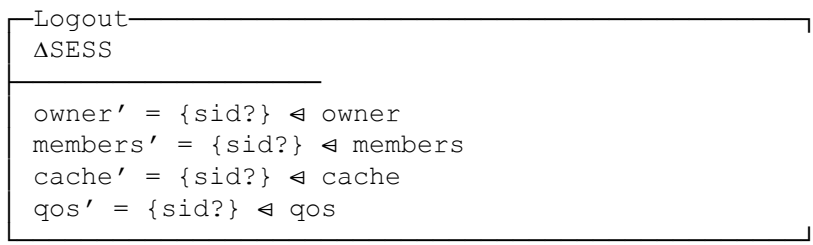
```
┌─Discard_Cached_Statement────────────────────────────────┐
│ ΔSESS                                                   │
│ a? : AUTY                                               │
│ stmt? : STMT                                            │
├─────────────────────                                    │
│ (a?,stmt?) ∈ cache sid?                                 │
│                                                         │
│ members'                                                │
│  = members ⊕ {sid? ↦ members sid? \ {stmt?.server}}      │
│ cache'                                                  │
│  = cache ⊕ {sid? ↦ cache sid? \ {(a?,stmt?)}}           │
│                                                         │
│ owner' = owner                                          │
│ qos' = qos                                              │
└─────────────────────────────────────────────────────────┘
```

(1) The Statement is in the Authority's cache for that Session.

(2) The Server is removed from the members of the Session, and the Statement is removed from the cache.

13.5 Ending a Session—Log-out

Log-out could be initiated by the Principal, or by an Authority, for example it could time-out.



The information about the Session, the owner, members, cached Statements and qos, is removed from the system.

14. Transactions

If a Server is in a Principal's Session, the Principal can make a request to the Server. The request might still be refused, if the Principal does not have the right to make that particular request, or if it is made at an inappropriate quality of service.

The request to a Server and the subsequent actions is known as a Transaction. A Transaction starts with the making of the request, and finishes when the request has been dealt with, or refused. It includes access control decisions.

```
[DECISION]
```

`DECISION` is the set of all access control decisions.

14.1 Multiple Principals

Whether a request made to a Server is carried out can depend on the access rights of more than one Principal. Consider the case where X has made a request to Y which requires Y to make a further request to Z. In the second request the Client is Y and the Server is Z, but the Principal could be X, or Y, or X and Y. If the sequence of requests gets longer, more combinations of Principals become possible.

When several Principals are involved in a request, it must be possible to distinguish them. For example, it might be necessary to know which Principal was initially responsible for the request, for accounting purposes.

14.2 Successful request

The decision whether to allow a request is described in Successful_Rqst. The process of making a decision does not change the state of the DCS.

A list of Principals, ps?, that own the respective Sessions sids?, make the request rqst? to Server s?, to be carried out at quality of service q?. This request is handled by the Client c?. The SLA, a?, makes the decision.

```
┌─Successful_Rqst──────────────────────────────────────┐
│ ΞDCS                                                 │
│ sids? : seq SID                                      │
│ ps? : seq ENTY                                       │
│ c?,s? : ENTY                                         │
│ a? : AUTY                                            │
│ rqst? : RQST                                         │
│ q? : QOS                                             │
│ decision! : DECISION                                 │
├──────────────────────                                 │
│ # sids? = # ps?                                      │
│ ran sids? ⊆ dom owner                                │
│                                                      │
│ ∀ n : ℕ | n ∈ 1 .. # ps? •                           │
│   ps? n = owner (sids? n) ∧ q? ≤ qos (sids? n)       │
│                                                      │
│ a? = local_auty s?                                   │
│ (s?,rqst?) ∈ enty_rqst q?                            │
│                                                      │
│ ∀ n : ℕ | n ∈ 1 .. # ps? • s? ∈ members (sids? n)    │
│                                                      │
│ ∀ n : ℕ | n ∈ 1 .. # ps? •                           │
│   ∃ stmt : STMT | (a?,stmt) ∈ cache (sids? n) •       │
│     ps? n = stmt.principal ∧ s? = stmt.server        │
│     ∧ rqst? ∈ stmt.requests                          │
│                                                      │
│ decision! = "Go ahead"                               │
└──────────────────────────────────────────────────────┘
```

(1) There are as many session ids as Principals, and each Session id refers to an existing Session.
(2) Each Principal in the list owns the corresponding Session. The quality of service for this request (the discretionary qos) must not be greater than any of the Sessions' qos; in most cases it will be the same.
(3) The deciding Authority is the SLA. The request must be a possible one for the Server at this qos.
(4) The Server must be in all the Principals' Sessions.
(5) For every Principal, the SLA has a cached statement relating to the Principal's Session that says the Principal is allowed to make that request. Again, since the Statements are in the SLA's cache, the SLA is guaranteed to trust them.

14.3 The complete decision

The failure condition schemas referred to below are straightforwardly derived from the `Successful_Rqst` schema.

The complete decision is

```
Decision ≙ Successful_Rqst
           ∨ No_Access_Right
           ∨ Not_in_all_Sessions
           ∨ Invalid_Request
           ∨ Auty_not_SLA
           ∨ Qos_too_High
           ∨ Not_owner
           ∨ Unknown_Sid
           ∨ Unknown_Server
```

This completes the formal description of the model.

15. Prolog Animation

In order to test and explore the Z specification, a Prolog animation was written. Since a Z specification is a collection of predicates, conversion to Prolog was relatively straightforward.

A few clauses had been written to handle named sets, functions and relations (based on those given in Reference 7). Z sets were implemented as Prolog lists. Since Z functions are sets of ordered pairs, they were implemented as lists of two-component lists, the first component being `x`, the second being the corresponding `y`:

```
{ (x1,y1),(x2,y2),(x3,y3) } ⟶ [ [x1,y1], [x2,y2], [x3,y3] ]
```

Once these low-level clauses had been written, an almost line-for-line translation of Z schemas to Prolog clauses was possible. For example, the log-in command (section 13.1), became in Prolog

```
login(P, S, QOS, SID) :- unsuccessful_login(P, S, QOS, SID).
login(P, S, QOS, SID) :- successful_login(P, S, QOS, SID).

unsuccessful_login(P, S, QOS, SID) :-
  enty(E), not(member(P,E)),
  sid = sid0,
  report(" Unknown Principal, login abandoned").

successful_login(P, S, QOS, SID) :-
  enty(E), member(P,E),
  gensym(sid, SID),
  override(owner, [ [SID, P] ]),
  override(members, [ [SID, [S]] ]),
  override(cache, [ [SID, []] ]),
  override(qos, [ [SID, QOS) ]),
  report("Login successful"),
  draw_login(P, QOS).
```

Most of the Prolog writing effort was spent on the animation's screen display (the purpose of the `report` and `draw_login` clauses above).

Also, sequencing information had to be included; decisions about what to do if a request failed had to be made. For example, although it would have been possible, on discovering there were no cached statements when trying a request, to have the Prolog report this and stop, it would have made a rather unimpressive demonstration! So in this case, an attempt would be made to gather the necessary statements, and retry the request.

This animation of the specification proved to be beneficial; a small bug in an earlier version of the model was exposed, and some consequences of the model were made explicit. Our confidence in the model was greatly increased. We are now going on to develop a full implementation for Project ADMIRAL.

16. Experience

Writing a version of the model in Z was a useful exercise. It forced us to think about the problem carefully, and this resulted in several changes in the model, some of them significant.

One area where the formalization had a big influence was in the property of Trust (sections 9 and 10). Although introduced in the original informal model, it remained a vague and woolly concept for a long time, despite attempts to pin it down. Everybody had their own subtly different idea about what was meant by Trust; this led to all sorts of problems.

This state of affairs was impossible in the Z version. Since it had to be formalized, various concrete properties had to be written down, or deliberately excluded. For example, it had been assumed, on and off, that Trust had to be transitive (A trusts B and B trusts C means A trusts C). As soon as this was formalized, it became apparent that it would be very easy for all authorities to end up trusting all the others, making the concept of Trust useless. So this property was removed.

In an earlier version of the model, Trust was simply a relation between authorities:

$$trusted\_by0 : AUTY \leftrightarrow AUTY$$

Either A trusts B, or it doesn't. The model was fairly well developed before it was decided this was a bit too liberal, and it would be better for Trust to apply to certain entities, allowing Authorities to keep tighter control over the particular resources. It was gratifying how straightforward the modification to

$$trusted\_by : AUTY \rightarrow (AUTY \leftrightarrow ENTY)$$

was. This gave us confidence that our model had a good structure.

17. Conclusions

The model's structure has improved, making it more understandable. This is of benefit to the potential users-the other ADMIRAL partners. Since it is clearer, they can feel happier that it meets their informal requirements for the system, or they can spot where it deviates, and make valuable criticisms. It also benefits the implementors. They can have more confidence that they understand what the designers intended.

An abstract, formal specification helps implementors in another way. For example, at this level the model contains no sequencing information, no specification of what to do if a request fails. Implementations require this information, and it would have to be added at the next stage in the development. But it seems appropriate not to have it in as a constraint at this level, leaving implementors free to choose the best strategy under different circumstances.

Various consequences of the choices made in the model can be deduced using the Z formalism, and certain desirable properties can be proved to hold. We hope to go on and produce some of these proofs as the next stage in the formal development.

The model itself provides a framework in which administrations can maintain autonomy, while allowing close interaction between the distributed systems. An implementation of this model is being developed for a real system-project ADMIRAL's access control facility.

Acknowledgements

We would like to thank the other participants of project ADMIRAL for their contributions, especially Nick Pope and Dave Robinson for many discussions. Jim Woodcock from Oxford Programming Research Group provided valuable help with the Z.

Appendix: Z Notation Used in this Paper

`x : T`	declaration of `x` as type `T`
`x : seq T`	`x` is a sequence of elements of type `T`
`LHS = RHS`	`LHS` is syntactically .equivalent to `RHS`
`P ∧ Q`	`P` and `Q`
`P ∨ Q`	`P` or `Q`
`∀ x : T • P`	for all x of type `T`, `P` holds
`∃ x : T • P`	there exists an `x` of type `T` such that `P`
`e ∈ S`	set membership
`e ∉ S`	`e` is not a member of `S`
`S ⊆ T`	`S` is a subset of `T`
`A × B`	Cartesian product
`ℙ S`	power set; set of all subsets of `S`
`A ∪ B`	set union
`⋃ S`	distributed set union
`A \ B`	set difference
`A ⟷ B`	set of relations from `A` to `B`; `= ℙ(A × B)`
`A ⇸ B`	set of partial functions from `A` to `B`
`A ⟶ B`	set of total functions from `A` to `B`
`x ↦ y`	maplet; `= (x,y)`
`dom R`	domain of relation `R`
`ran R`	range of relation `R`
`A ⩤ R`	domain subtraction; `dom (A ⩤ R) = dom R \ A`
`R ⊕ S`	function overriding; `= (domS ⩤ R) ∪ S`
`ℕ`	`= {0,1,2,3,. . . }`
`#s`	length of sequence `s`
`h ⊢ P`	`P` is a theorem, given hypothesis `h`
`S \ (x)`	The schema `S` with variable `x` hidden: `x` is removed from the signature and existentially quantified in the predicate.

References

1. N. H. Pope D. J. Tolcher, S. R. Wilbur and R. A. Rosner, 'Project, ADMIRAL — research into networks and distributed systems', *Networks* 86, Online Publications, Middlesex, England, 1986.
2. S. P. Lord, N. H. Pope and S. Stepney, 'Access management in multi-administration networks', *IEE Secure Communication Systems*, 1986.
3. C. E. Landwehr, 'Formal models for computer security', *Computing Surveys*, **13**(3), 247-278 (1981).
4. M. H. Cheheyl, M. Gasser, G. A. Huff and J. K. Millen, 'Verifying security', *Computing Surveys*, **13**(3), 279-339 (1981).

S. C. Morgan and B. Sufrin, 'Specification of the Unix filing system', *IEEE Trans. Softw. Eng.*, **SE-10**(2), 128-142 (1984).

6. B. Sufrin, C. Morgan, I. Sorensen and I. Hayes, *The Z Handbook*, Programming Research Group, Oxford, 1984.
7. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer, 1984.