Where IT all begins

InterNational Committee for Information Technology Standards (INCITS)
Secretariat: Information Technology Industry Council (ITI)
1101 K Street NW, Suite 610, Washington, DC 20005
www.INCITS.org

eb-2014-00267

Document Date:	5/2/2014
To:	INCITS Members
Reply To:	Rachel Porter
Subject:	Public Review and Comments Register for the Approval of:
	INCITS 526-201x, Information technology - Next Generation Access Control - Generic Operations and Data Structures (NGAC-GOADS)
Due Date:	**The public review is from May 16, 2014 – June 30, 2014.**

Action:

The InterNational Committee for Information Technology Standards (INCITS) announces that the subject-referenced document(s) is being circulated for a 45-day public review and comment period. Comments received during this period will be considered and answered. Commenters who have objections/suggestions to this document should so indicate and include their reasons.

All comments should be forwarded not later than the date noted above to the following address:

INCITS Secretariat/ITI
1101 K Street NW - Suite 610
Washington DC 20005-3922
Email: comments@itic.org (preferred)

This public review also serves as a call for patents and any other pertinent issues (copyrights, trademarks). Correspondence regarding intellectual property rights may be emailed to the INCITS Secretariat at patents@itic.org.

Working Draft American National Standard

Project CS1/2195-D

Revision 1.30
11 April 2014

Information technology - Next Generation Access Control - Generic Operations and Data Structures (NGAC-GOADS)

CS1 Technical Editor: Wayne Jansen
Booz Allen & Hamilton
8283 Greensboro Drive
McLean, VA 22102

USA

Telephone: 703-377-0375
Email: Jansen_Wayne@bah.com

Reference number
ANSI INCITS.*:200x**

Points of Contact

InterNational Committee for Information Technology Standards (INCITS) CS1 Technical Committee

CS1 Chair
Dan Benigni
NIST
Stop 8930
Gaithersburg, MD 20899-8930
USA

Telephone: (301) 975-3279
Email: dbegnini@nist.gov

CS1 Vice-Chair
Sal Francomacaro
NIST
Stop 8930
Gaithersburg, MD 20899-8930
USA

Telephone: (301) 975-6414
Email: salvatore.francomacaro@nist.gov

CS1 Web Site: http://cs1.incits.org/

CS1 E-mail reflector: cyber-security@standards.incits.org

INCITS Secretariat
Suite 200
1250 Eye Street, NW
Washington, DC 20005
USA

Telephone: 202-737-8888
Web site: http://www.incits.org
Email: incits@itic.org

Information Technology Industry Council
Web site: http://www.itic.org

Document Distribution
INCITS Online Store
managed by Techstreet
1327 Jones Drive
Ann Arbor, MI 48105
USA

Web site: http://www.techstreet.com/incits.html
Telephone: (734) 302-7801 or (800) 699-9277

Revision Information

Version 0.10 (23 July, 2010)
Initial Draft

Version 1.20 (31 January, 2014)
Draft for Letter Ballot

Version 1.30 (11 April 2014)
Corrected Letter Ballot Draft: Fixed noted typographical errors, access request parameter inconsistencies and minor faults in semantic expressions.

American National Standards
for Information Systems

Next Generation Access Control - Generic Operations and Data Structures (NGAC-GOADS)

Secretariat

InterNational Committee for Information Technology Standards

Approved mm.dd.yy

American National Standards Institute, Inc.

Abstract

Next Generation Access Control (NGAC) is a fundamental reworking of traditional access control into a form that suits the needs of the modern distributed interconnected enterprise. NGAC is based on a flexible infrastructure that can provide access control services for a number of different types of resources, and when they are accessed by a number of different types of application and user. That infrastructure is scalable, able to support policies of different types simultaneously, and remain manageable in the face of almost constant change. This standard contains a complete, detailed description of the definitions and abstractions needed to realize the architecture defined by the NGAC-FA standard. It details a fixed set of configurable data relations and a fixed set of functions that are capable of expressing and specifying a wide range of different types of access control policies of a wide range of complexities.

Published by
American National Standards Institute
11 W. 42nd Street, New York, New York 10036

Table of Contents

Topic **Page**

List of Figures

Foreword

(This foreword is not part of American National Standard INCITS.***:200x.)

Technical Committee CS1 of Accredited Standards Committee INCITS developed this standard during 2012-2013. The standards approval process started in 2014.

Next Generation Access Control (NGAC) is a fundamental reworking of traditional access control to a form that suits the needs of the modern distributed interconnected enterprise. NGAC is based on a flexible infrastructure that can provide access control services for a number of different types of resources, and when they are accessed by a number of different types of applications and users. The NGAC infrastructure is scalable, able to support policies of different types simultaneously, and remains manageable in the face of changing technology, organizational restructuring, and increasing data volumes.

This standard contains a complete, detailed description of the definitions and abstractions needed to realize the architecture defined by the NGAC-FA standard. The abstractions are based on the mathematics of set theory and predicate calculus to provide a precise specification. By capturing the essential properties of NGAC, free from constraints on how these properties are achieved, this standard serves as a conceptual model for the design and implementation of NGAC.

This standard contains the following items:

a) detailed specifications of the abstract data structures needed to support the definitions in NGAC-FA;
b) a detailed description of how the abstract structures in a) above are used to represent an access control policy and form authorization decisions;
c) detailed descriptions of the generic commands needed to support the administration access information flows in NGAC-FA, in terms of the descriptions in b) above;
d) an informative annex containing an example of using formal grammars to specify the pattern and response components of an obligation;
e) an informative annex containing examples of representing common access control methods such as Role-Based Access Control (RBAC) and Chinese Wall in terms of a), b), c) and d) above;
f) an informative annex of references; and
g) a normative annex summarizing the mathematical notation used in this standard.

Requests for interpretation, suggestions for improvement and addenda, or defect reports are welcome. They should be sent to the INCITS Secretariat, InterNational Committee for Information Technology Standards, Information Technology Institute, 1250 Eye Street, NW, Suite 200, Washington, DC 20005-3922.

Users of this standard are encouraged to determine if there are standards in development or new versions of this standard that may extend or clarify technical information contained in this standard.

This standard was processed and approved for submittal to ANSI by the InterNational Committee for Information Technology Standards (INCITS). Committee approval of the standard does not necessarily imply that all committee members voted for approval. At the time of it approved this standard, INCITS had the following members:

Organization Represented	**Name of Representative**

Editor’s Note 1: <<Insert INCITS member list>>

Introduction

This standard is divided into these clauses and annexes:

Clause 1 defines the scope of this standard and places it in context of other standards and standards projects.

Clause 2 enumerates the normative references that apply to this standard.

Clause 3 describes the definitions, symbols, abbreviations, and notation used in this standard.

Clause 4 defines the abstract data structures that are the basis by which policy governing resource and administrative operations is defined in NGAC.

Clause 5 specifies the generic commands needed to support the administration access information flows in NGAC-FA.

Annex A provides example formal descriptions of the pattern and response components of an obligation.

Annex B provides examples of representing existing real-world approaches to access control using the facilities of NGAC.

Annex C lists the set of references used in the examples given in annex B.

Annex D summarizes the mathematical notation used in this standard, which corresponds to a subset of the Z formal specification notation defined in ISO/IEC 13568:2002.

AMERICAN NATIONAL STANDARD **BSR INCITS.***:200x**

American National Standard
for Information Technology -

Next Generation Access Control - Generic Operations and Data Structures (NGAC-GOADS)

1 Scope

Next Generation Access Control (NGAC) is a fundamental reworking of traditional access control into a form that suits the needs of the modern distributed interconnected enterprise. The NGAC family of standards provides the architectural, functional and interface definitions necessary to create an effective access control system.

Access control is both an administrative and an automated process of defining and restricting which users and their processes can perform which operations on which system resources. The information that provides the basis by which access requests are granted or denied is known as a security policy. A security model is formal representation of a security policy and its working. A wide variety of policies types and supporting security models have been created to address different situations. Well-known examples of mechanisms by which specific policy types are enforced are access control lists (ACLs), capabilities, role based access control (RBAC), and type enforcement, and well-known policies are discretionary access control (DAC), RBAC, multi-level security (MLS) and Chinese Wall.

NGAC diverges from traditional approaches to access control in defining a generic architecture that is separate from any particular policy or type of policy. NGAC is not an extension of, or adaption of, any existing access control mechanism, but instead is a redefinition of access control in terms of a fundamental and reusable set of data abstractions and functions. NGAC provides a unifying framework capable without extension of supporting not only current access control approaches, but also novel types of policy that have been conceived but never implemented due to the lack of a suitable enforcement mechanism.

NGAC accommodates combinations of different policies merely by changes to its control information, and thus it is possible to have several types of policy policies supported concurrently in a manner that is both deterministic and manageable. NGAC is particularly suitable for applications in which some information is stored locally and some is stored in a grid or cloud, since different policies can be asserted in each situation. Even more generally, NGAC supports a situation where a formal policy determined by a central organization is combined with a local, specific and more ad-hoc policy required to meet local needs.

In addition to its support of policies, NGAC also enables support for a variety of data services, including e-mail, workflow, records management etc. Support for these services is established through NGAC control information contained in a database within NGAC.

The set of NGAC standards specifies the architecture, functions, operations, and interfaces necessary to ensure interoperability between conforming NGAC implementations. This standard defines generic operations and data structures. Conforming implementations may employ any design technique that does not violate interoperability.

2 Normative References

2.1 Normative references

The following standards contain provisions that, by reference in the text, constitute provisions of this standard. At the time of publication, the editions listed were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below.

Copies of the following documents may be obtained from ANSI, an ISO member organization:

Approved ANSI standards;
approved international and regional standards (ISO and IEC); and
approved foreign standards (including JIS and DIN).

For further information, contact the ANSI Customer Service Department:

Phone:	+1 212-642-4900
Fax:	+1 212-302-1286
Web:	http://www.ansi.org
E-mail:	ansionline@ansi.org

or the InterNational Committee for Information Technology Standards

(INCITS):	Phone 202-626-5738
Web:	http://www.incits.org
E-mail:	incits@itic.org

Additional availability contact information is provided below as needed.

2.2 Approved references

NGAC-FA: INCITS 499-2013, *American National Standard for Information Technology – Next Generation Access Control - Functional Architecture (NGAC-FA)*

ZNOT: ISO/IEC 13568:2002, *Information technology – Z formal specification notation – Syntax, type system and semantics*

2.3 References under development

At the time of publication, the following referenced American National Standards were still under development. For information on the current status of the document, or regarding availability, contact the relevant standards body or other organization as shown.

NGAC-IRPADS: INCITS Project 2193-D, *Information technology – Next Generation Access Control - Implementation Requirements, Protocols and API Definitions (NGAC-IRPADS)*

3 Definitions, Symbols, Abbreviations, and Conventions

3.1 Definitions

For the purposes of this document, the terms and definitions in NGAC-FA apply, as do the following terms and definitions.

Access right: A property that must be held by a user to perform operations on information persisted in the PIP and on object resources.

Access right set: A set of access rights that are related for the purposes of access control.

Access request: The information issued by a process on behalf of a user, which specifies a sequence of arguments together with an operation to be performed using the argument sequence.

Authorization decision: The result of evaluating an access request with respect to authorizations defined with a configured NGAC policy.

Authorization state: The basic elements and containers, together with the relations that define the prevailing access rights between these entities.

Policy elements: The users, user attributes, object attributes and policy classes associated with a particular access control policy.

Policy element diagram: The directed graph representing all policy elements and the assignment relation over them.

Referent: A policy element used in NGAC relations to represent the section of the policy element diagram rooted at the policy element.

3.2 Symbols and abbreviations

Symbol / Abbreviation	Meaning (see NGAC-FA for further information)
API	Application Programming Interface
DAC	Discretionary Access Control
EPP	Event Processing Point
NGAC	Next Generation Access Control
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
RBAC	Role-Based Access Control
URG	Unprotected Resource Gate

3.3 Keywords

Invalid: A keyword used to describe an illegal or unsupported bit, byte, word, field or code value. Receipt of an invalid bit, byte, word, field or code value shall be reported as an error.

Mandatory: A keyword indicating an item that is required to be implemented as defined in this standard to claim compliance with this standard.

May: A keyword that indicates flexibility of choice with no implied preference.

May not: Keywords that indicates flexibility of choice with no implied preference.

Optional: A keyword that describes features that are not required to be implemented by this standard. However, if any optional feature defined by this standard is implemented, it shall be implemented as defined in this standard.

Reserved: A keyword referring to bits, bytes, words, fields and code values that are set aside for future standardization. Their use and interpretation may be specified by future extensions to this or other standards. A reserved bit, byte, word or field shall be set to zero, or in accordance with a future extension to this standard. Recipients are not required to check reserved bits, bytes, words or fields for zero values. Receipt of reserved code values in defined fields shall be reported as an error.

Shall: A keyword indicating a mandatory requirement. Designers are required to implement all such requirements to ensure interoperability with other products that conform to this standard.

Should: A keyword indicating flexibility of choice with a preferred alternative; equivalent to the phrase "it is recommended".

3.4 Conventions

Certain words and terms used in this American National Standard have a specific meaning beyond the normal English meaning. These words and terms are defined either in clause 3 or in the text where they first appear.

Numbers immediately followed by lower-case b (xxb) are binary values.

Numbers that are not immediately followed by lower-case b are decimal values.

An alphabetic list (e.g., a, b, c) or a bulleted list of items indicate the items in the list are unordered.

A numeric list (e.g., 1, 2, 3) of items indicate the items in the list are ordered (i.e., item 1 shall occur or complete before item 2).

The mathematical notation used in GOADS corresponds to a subset of the Z formal specification notation defined in ISO/IEC 13568:2002, but is specific to this work. Annex C provides a summary of the notation.

In the event of conflicting information, the precedence for requirements defined in this standard is as follows:

1) formal notation,
2) text, then
3) figures.

4 Abstract Data Structures

4.1 Overview

This standard defines the abstract data structures that govern the operation of the NGAC Functional Architecture (see NGAC-FA). These structures are used to represent security policies and provide the basis for rendering authorization decisions. The types of structures that are defined in this subclause are as follows:

a) the sets of element identifiers that are maintained by the PIP to represent the basic elements (see 4.2) that are the fundamental entities in which operation of the NGAC framework is described;
b) the different types of containers (see 4.3) maintained by the PIP to represent the characteristics of basic elements and to identify basic elements that share those characteristics; and
c) the various relations and functions (see 4.4) that are maintained by the PIP to represent configured relationships among the basic elements and containers, and form the basis of the policies enforced by the NGAC functional architecture.

4.2 Basic elements

4.2.1 Background

The NGAC element specifications define the abstract data structures that correspond to the basic elements and properties defined in NGAC-FA. They are described in this subclause as follows:

a) users (see 4.2.2);
b) processes (see 4.2.3);
c) objects (see 4.2.4);
d) operations (see 4.2.5); and
e) access rights (see 4.2.6).

4.2.2 Users

NGAC users shall be represented by a finite set of user element identifiers.

$U = \{u_1, \ldots , u_n\}$

4.2.3 Processes

NGAC processes shall be represented by a finite set of process element identifiers.

$P = \{p_1, \ldots , p_n\}$

4.2.4 Objects

NGAC objects shall be represented by a finite set of object element identifiers.

$O = \{o_i, \ldots , o_n\}$

4.2.5 Operations

NGAC generic operations shall be represented by a finite set of operation identifiers.

$Op = \{op_i, \dots , op_n\}$

Generic operations are partitioned into two distinct, finite sets of operations: resource operations and administrative operations. Resource operations are used for access to protected resources, while administrative operations are used for access to data structures and information persisted in the PIP.

$ROp = \{rop_1, \dots , rop_n\}$
$AOp = \{aop_1, \dots , aop_n\}$
$Op = ROp \cup AOp$

4.2.6 Access rights

Access rights shall be represented by a finite set of access right identifiers. One or more access rights may be required to carry out an operation.

$AR = \{ar_1, \dots , ar_n\}$

4.3 Containers

4.3.1 Background

NGAC containers represent common characteristics and properties of basic elements. They are described in this subclause as follows:

a) user attributes (see 4.3.2);
b) object attributes (see 4.3.3); and
c) policy classes (see 4.3.4).

4.3.2 User attributes

User attributes shall be represented by a finite set of user attribute identifiers.

$UA = \{ua_1, \dots , ua_n\}$

4.3.3 Object attributes

Object attributes shall be represented by a finite set of object attribute identifiers.

$OA = \{oa_1, \dots oa_n\}$

Every member of the set O is by definition also a member of OA.

$O \subseteq OA$

4.3.4 Policy classes

Policy classes shall be represented by a finite set of policy class identifiers.

$PC = \{pc_1, \dots , pc_n\}$

4.4 Relations

4.4.1 Background

NGAC configured relations are a part of the information persisted in a PIP that are managed through administrative actions, and determine the configuration of an NGAC Functional Architecture. NGAC relations are either configured relations or derived relations. NGAC configured relations represent relationships among basic elements and/or containers. NGAC derived relations are calculated from configured relations for the purpose of rendering an access control decision or conducting an administrative review. The following NGAC configured relations are described in this subclause:

a) assignment (see 4.4.2);
b) association (see 4.4.3);
c) prohibition (see 4.4.4); and
d) obligation (see 4.4.5).

4.4.2 Assignment

The NGAC policy elements used in assignments comprise the set of all users, user attributes, object attributes (which include all objects) and policy classes; the policy element set is defined as follows:

$PE = U \cup UA \cup OA \cup PC$

The assignment relation is a binary relation on the set PE, which has the following properties:

a) It is irreflexive;
b) It is acyclic;
c) It is policy class connected (i.e., A sequence of assignments exists from every element of (PE \ PC) to an element of PC); and
d) It precludes object attribute to object assignments.

The assignment relation is formally defined as follows:

$ASSIGN \subseteq (U \times UA) \cup (UA \times UA) \cup (OA \times OA) \cup (UA \times PC) \cup (OA \times PC)$, where the following properties hold:

$\forall x, y \in PE$: $((x, y) \in ASSIGN \Rightarrow x \neq y) \land$
$\nexists s \in iseq_1\ PE$: $(\#s > 1 \land \forall i \in \{1,...,(\#s - 1)\}: ((s(i), s(i+1)) \in ASSIGN) \land$
$(s(n), s(1)) \in ASSIGN) \land$
$\forall w \in (PE \setminus PC), \exists s \in iseq_1\ PE$: $(s(1) = w \land s(n) \in PC \land \forall i \in \{1,...,(\#s - 1)\}: ((s(i), s(i+1)) \in ASSIGN)) \land$
$\forall x \in OA, \forall y \in PE$: $((x, y) \in ASSIGN \Rightarrow y \notin O)$

The assignment relation can be represented as a directed graph or digraph G = (PE, ASSIGN), where PE are the vertices of the graph, and each tuple (x, y) of ASSIGN represents a direct edge or arc that originates at x and terminates at y. A digraph of policy elements and the assignments among them is referred to as a policy element diagram.

An object can be said to be “contained by” an object attribute if a sequence of assignments that links the object with the object attribute exists. The Objects function represents the mapping from an object attribute to the set of objects that are contained by that object attribute. Intuitively, the function Objects(oa) returns the set of objects that are contained by or possess the characteristics of the object attribute oa. The Objects function is a total function from OA to 2^O, which is defined as follows:

Objects $\subseteq$ OA $\times$ 2^{O}, where the following properties hold:

$\forall$oa $\in$ OA, $\forall$x $\in$ 2^{O}: ((oa, x) $\in$ Objects $\Leftrightarrow$ ($\forall$o $\in$ O: (o $\in$ x $\Leftrightarrow$ (o, oa) $\in$ ASSIGN*)))

Similarly, the Users function is a total function from UA to 2^{U}, which represents the mapping from a user attribute to the set of users that are contained by that user attribute. The Users function is defined as follows:

Users $\subseteq$ UA $\times$ 2^{U}, where the following properties hold:

$\forall$ua $\in$ UA, $\forall$x $\in$ 2^{U}: ((ua, x) $\in$ Users $\Leftrightarrow$ ($\forall$u $\in$ U: (u $\in$ x $\Leftrightarrow$ (u, ua) $\in$ $ASSIGN^{+}$)))

The containment concept can also be generalized for any policy element. The Elements function is a total function from PE to 2^{PE}, which represents the mapping from a given policy element to the set of policy elements that includes the policy element and all the policy elements contained by that policy element. The Elements function is defined as follows:

Elements $\subseteq$ PE $\times$ 2^{PE}, where the following properties hold:

$\forall$pe $\in$ PE, $\forall$x $\in$ 2^{PE}: ((pe, x) $\in$ Elements $\Leftrightarrow$ ($\forall$e $\in$ PE: (e $\in$ x $\Leftrightarrow$ (e, pe) $\in$ ASSIGN*)))

A policy element can also be viewed as a referent or representative for the entire section of the policy element diagram rooted at the policy element. Stated more formally, for the policy element diagram G = (PE, ASSIGN), a referent r $\in$ PE represents the subgraph G' = (PE', ASSIGN'), where PE' = Elements(r) and ASSIGN' = {(x, y) | x, y $\in$ PE' $\wedge$ (x, y) $\in$ ASSIGN}. A policy element when viewed as a referent serves as a designator for not only itself, but also for policy elements contained by the referent.

4.4.3 Association

4.4.3.1 Background

An association relation is a ternary relation from UA to (2^{AR} \ {∅}) to PE. It is defined as follows:

ASSOCIATION $\subseteq$ UA $\times$ (2^{AR} \ {∅}) $\times$ PE

Three other relations are derived from an association and are described in this subclause as follows:

1) privilege (see 4.4.3.2);
2) capability (see 4.4.3.3); and
3) access control entry (ACE) (see 4.4.3.4).

4.4.3.2 Privilege

A privilege relation involves a user, access right and policy element. It is derived from one or more association relations. For each policy class that contains the policy element, an association relation must exist such that the following is true:

a) the user is contained by the user attribute of the association;
b) the policy element in question is contained by the policy element of the association;
c) the policy element of the association is contained by the policy class; and;
d) the access right is a member of the access right set of the association.

The privilege relation is a ternary relation from U to AR to PE. It is formally defined as follows:

PRIVILEGE $\subseteq$ U×AR×PE, where the following properties hold:

$\forall u\in U, \forall ar\in AR, \forall pe\in pe$: ((u, ar, pe) $\in$ PRIVILEGE $\Leftrightarrow \forall pc\in PC$: ((pe, pc) $\in$ ASSIGN$^+$ $\Rightarrow$
$\exists ua\in UA, \exists ars\in 2^{AR}, \exists x\in PE$: ((ua, ars, x) $\in$ ASSOCIATION $\wedge$
u $\in$ Users(ua) $\wedge$ ar $\in$ ars $\wedge$ pe $\in$ Elements(x) $\wedge$ x $\in$ Elements(pc))))

4.4.3.3 Capability

A capability represents actions that can potentially be taken against policy elements, including objects, barring any prohibitions. The capability relation is derived from association relations via privileges and is defined as follows:

CAPABILITY $\subseteq$ AR×PE, where $\forall ar\in AR, \forall pe\in PE$: ((ar, pe) $\in$ CAPABILITY $\Leftrightarrow \exists u\in U$: (u, ar, pe) $\in$ PRIVILEGE)

A user has a capability described by the relation, if and only if a privilege exists specifying that user, access right and policy element.

4.4.3.4 Access Control Entry

An access control entry represents actions that users can potentially take, barring any prohibitions. An Access Control Entry (ACE) relation is derived from association relations via privileges, and is defined as follows:

ACE $\subseteq$ U×AR, where $\forall u\in U, \forall ar\in AR$: ((u, ar) $\in$ ACE $\Leftrightarrow \exists pe\in PE$: (u, ar, pe) $\in$ PRIVILEGE)

A policy elements has the ACE described by the relation, if and only if a privilege exists specifying that user, access right and policy element.

4.4.4 Prohibition

4.4.4.1 Background

Three distinct, but related types of prohibition relations exist: user-based prohibitions (see 4.4.4.2), process-based prohibitions (see 4.4.4.3) and attribute-based prohibitions (see 4.4.4.4). These prohibitions denote an effective set of privileges that a specific user, process, or group of users is precluded from exercising, regardless of whether any of the privileges involved actually can or cannot be derived for the user, process or users in question.

The set of policy elements affected by a prohibition is designated via either conjunctive or disjunctive mappings over sets of referent policy elements to the policy elements in question. More precisely, the disjunctive range function represents the mapping from two constraint sets of policy elements, the first designating policy elements for inclusion and the second designating policy elements for exclusion, to a set of policy elements formed by logical disjunction of the policy elements contained within or not contained within the subgraphs of the referent policy elements of each constraint set respectively. More precisely, the set of policy elements returned by the disjunctive range function, DisjRange(peis, pees), where peis$\in 2^{PE}$ and pees$\in 2^{PE}$, is the union of Elements(pei), for all pei in the inclusion set peis, along with the union of (PE \ Elements(pee)), for all pee in the exclusion set pees, which can be more succinctly expressed as follows:

$$\text{DisjRange(peis, pees)} = \bigcup_{pei\in peis} \text{Elements(pei)} \cup \bigcup_{pee\in pees} (PE \setminus \text{Elements(pee)})$$

The disjunctive range function is a total binary function from $2^{PE} \times 2^{PE}$ to 2^{PE}, and is formally defined as follows:

DisjRange $\subseteq (2^{PE} \times 2^{PE}) \times 2^{PE}$, where the following properties hold:

$\forall peis \in 2^{PE}, \forall pees \in 2^{PE}, \forall pes \in 2^{PE}$: $(pes \in DisjRange(peis, pees) \Leftrightarrow (\forall pei \in peis, \forall pee \in pees, \forall pe \in pes: (pe \in Elements(pei) \lor pe \in (PE \setminus Elements(pee)))))$

Similarly, the conjunctive range function represents the mapping from two constraint sets of policy elements, the first designating policy elements for inclusion and the second designating policy elements for exclusion, to a set of policy elements formed by logical conjunction of the policy elements contained by or not contained by the policy elements of each constraint set respectively. More precisely, the set of policy elements returned by the conjunctive range function, ConjRange(peis, pees), where $peis \in 2^{PE}$ and $pees \in 2^{PE}$, is the intersection of Elements(pei), for all pei in the inclusion set peis, along with the intersection of (PE \ Elements(pee)), for all pee in the exclusion set pees, which can be more succinctly expressed as follows:

$$ConjRange(peis, pees) = \bigcap_{pei \in peis} Elements(pei) \cap \bigcap_{pee \in pees} (PE \setminus Elements(pee))$$

The conjunctive range function is a total binary function from $2^{PE} \times 2^{PE}$ to 2^{PE}. The function is formally defined as follows:

ConjRange $\subseteq (2^{PE} \times 2^{PE}) \times 2^{PE}$, where the following properties hold:

$\forall peis \in 2^{PE}, \forall pees \in 2^{PE}, \forall pes \in 2^{PE}$: $(pes \in ConjRange(peis, pees) \Leftrightarrow (\forall pei \in peis, \forall pee \in pees, \forall pe \in pes: (pe \in Elements(pei) \land pe \in (PE \setminus Elements(pee)))))$

4.4.4.2 User-based prohibitions

User-based prohibition relations involve a quaternary relation from U to 2^{AR} to 2^{PE} to 2^{PE}, where the first set represents all users, the second set represents all access rights sets, and the third and fourth sets represent respectively all inclusion and all exclusion sets of policy elements. Two variants of user-based prohibition relations exist: a disjunctive and a conjunctive form. A disjunctive user prohibition tuple denotes that all processes initiated by the user are withheld the right to exercise any of the access rights defined in the access right set against any policy elements that lie within the disjunctive range of the inclusion and exclusion sets of policy elements. The access right set cannot be the empty set, and the inclusion and exclusion sets cannot both be the empty set. The relation is formally defined as follows:

U_DENY_DISJ $\subseteq U \times 2^{AR} \times 2^{PE} \times 2^{PE}$, where $\forall u \in U, \forall ars \in 2^{AR}, \forall peis \in 2^{PE}, \forall pees \in 2^{PE}$:
$((u, ars, peis, pees) \in U\_DENY\_DISJ \Rightarrow (ars \neq \emptyset \land peis \cup pees \neq \emptyset))$

Similarly, the conjunctive user prohibition tuple denotes that all processes initiated by the user are withheld the right to exercise any of the access rights defined in the access right set against any policy elements that lie within the conjunctive range of the inclusion and exclusion sets of policy elements. The relation is formally defined as follows:

U_DENY_CONJ $\subseteq U \times 2^{AR} \times 2^{PE} \times 2^{PE}$, where $\forall u \in U, \forall ars \in 2^{AR}, \forall peis \in 2^{PE}, \forall pees \in 2^{PE}$:
$((u, ars, peis, pees) \in U\_DENY\_CONJ \Rightarrow (ars \neq \emptyset \land peis \cup pees \neq \emptyset))$

A user restriction relation is derived from one or more user prohibition relations. It consists of a tuple of three parts: a user, access right and policy element, such that the following is true:

a) the user is designated by the user identifier of the prohibition;

b) the access right is a member of the access right set of the prohibition; and
c) the policy element lies within either the disjunctive or conjunctive range of the inclusion and exclusion policy element sets of the prohibition, respective of whether the prohibition is a disjunctive or conjunctive variant.

The user restriction relation is a ternary relation from U to AR to PE. It is formally defined as follows:

U_RESTRICT $\subseteq$ U×AR×PE, where the following properties hold:

$\forall u \in U, \forall ar \in AR, \forall pe \in PE$: ((u, ar, pe) $\in$ U_RESTRICT $\Leftrightarrow \exists ars \in 2^{AR}, \exists peis \in 2^{PE}, \exists pees \in 2^{PE}$:
(((u, ars, peis, pees) $\in$ U_DENY_DISJ $\land$ ar $\in$ ars $\land$ pe $\in$ DisjRange(peis, pees)) $\lor$
((u, ars, peis, pees) $\in$ U_DENY_CONJ $\land$ ar $\in$ ars $\land$ pe $\in$ ConjRange(peis, pees)))

4.4.4.3 Process-based prohibitions

A user must perform accesses indirectly through one or more processes that carry out actions on its behalf. Each process is related to only one user. The process-to-user mapping is a function from the domain P to the codomain U. It is formally defined as follows:

Process_User $\subseteq$ P×U, where $\forall p \in P, \exists_1 u \in U$: u = Process_User(p)

Process-based prohibitions are defined similarly to user-based prohibitions. A process-based prohibition relation is a quaternary relation from P to 2^{AR} to 2^{PE} to 2^{PE}. Similar to user-based prohibitions, the first set represents all processes, the second set represents all access rights sets, and the third and fourth sets represent respectively all inclusion and all exclusion sets of policy elements. Both disjunctive and conjunctive variants of process-based prohibition relations exist. A disjunctive process prohibition tuple denotes that the process is prohibited from exercising any of the access rights defined in the access right set against any of the policy elements that lie within the disjunctive range of the inclusion and exclusion policy element sets. The access right set cannot be the empty set, and the inclusion and exclusion sets cannot both be the empty set. The relation is defined as follows:

P_DENY_DISJ $\subseteq$ P×2^{AR}×2^{PE}×2^{PE}, where $\forall p \in P, \forall ars \in 2^{AR}, \forall peis \in 2^{PE}, \forall pees \in 2^{PE}$:
((p, ars, peis, pees) $\in$ P_DENY_DISJ $\Rightarrow$ (ars $\neq \emptyset$ $\land$ peis $\cup$ pees $\neq \emptyset$))

Similarly, the conjunctive process prohibition tuple denotes that the process is prohibited from exercising any of the access rights defined in the access right set against any of the policy elements that lie within the conjunctive range of the inclusion and exclusion policy element sets. The relation is defined as follows:

P_DENY_CONJ $\subseteq$ U×2^{AR}×2^{PE}×2^{PE}, where $\forall p \in P, \forall ars \in 2^{AR}, \forall peis \in 2^{PE}, \forall pees \in 2^{PE}$:
((p, ars, peis, pees) $\in$ P_DENY_CONJ $\Rightarrow$ (ars $\neq \emptyset$ $\land$ peis $\cup$ pees $\neq \emptyset$)

A process restriction relation is derived from one or more process prohibition relations. It consists of a tuple of three parts: a process, access right, and policy element, such that the following is true:

a) the process is designated by the process identifier of the prohibition;
b) the access right is a member of the access right set of the prohibition; and
c) the policy element lies within either the disjunctive or conjunctive range of the inclusion and exclusion policy element sets of the prohibition, respective of whether the prohibition is a disjunctive or conjunctive variant.

The process restriction relation is a ternary relation from P to AR to PE. It is formally defined as follows:

P_RESTRICT $\subseteq$ P×AR×PE, where the following properties hold:

$\forall p \in P, \forall ar \in AR, \forall pe \in PE$: $((p, ar, pe) \in P\_RESTRICT \Leftrightarrow \exists ars \in 2^{AR}, \exists peis \in 2^{PE}, \exists pees \in 2^{PE}$:
$(((p, ars, peis, pees) \in P\_DENY\_DISJ \land ar \in ars \land pe \in DisjRange(peis, pees)) \lor$
$((p, ars, peis, pees) \in P\_DENY\_CONJ \land ar \in ars \land pe \in ConjRange(peis, pees)))$

4.4.4.4 Attribute-based prohibitions

Attribute-based prohibitions are defined similarly to user-based prohibitions. An attribute-based prohibition relation is a quaternary relation from UA to 2^{AR} to 2^{PE} to 2^{PE}. Similar to user-based prohibitions, the first set represents all user attributes, the second set represents all access rights sets, and the third and fourth sets represent respectively all inclusion and all exclusion sets of policy elements. Both disjunctive and conjunctive variants of attribute-based prohibition relations exist. A disjunctive attribute prohibition tuple denotes that all processes initiated by any user contained by the attribute are prohibited from exercising any of the access rights defined in the access right set against any of the policy elements that lie within the disjunctive range of the inclusion and exclusion policy element sets. The access right set cannot be the empty set, and the inclusion and exclusion sets cannot both be the empty set. The relation is defined as follows:

$UA\_DENY\_DISJ \subseteq UA \times 2^{AR} \times 2^{PE} \times 2^{PE}$, where $\forall ua \in UA, \forall ars \in 2^{AR}, \forall peis \in 2^{PE}, \forall pees \in 2^{PE}$:
$((ua, ars, peis, pees) \in UA\_DENY\_DISJ \Rightarrow (ars \neq \emptyset \land peis \cup pees \neq \emptyset))$

Similarly, the conjunctive attribute prohibition tuple denotes that all processes initiated by any user contained by the attribute are prohibited from exercising any of the access rights defined in the access right set against any of the policy elements that lie within the conjunctive range of the inclusion and exclusion policy element sets. The relation is defined as follows:

$UA\_DENY\_CONJ \subseteq U \times 2^{AR} \times 2^{PE} \times 2^{PE}$, where $\forall ua \in UA, \forall ars \in 2^{AR}, \forall peis \in 2^{PE}, \forall pees \in 2^{PE}$:
$((ua, ars, peis, pees) \in UA\_DENY\_CONJ \Rightarrow (ars \neq \emptyset \land peis \cup pees \neq \emptyset)$

An attribute restriction relation is derived from one or more attribute prohibition relations. It consists of a tuple of three parts: an attribute, access right, and policy element, such that the following is true:

a) the attribute is designated by the user attribute identifier of the prohibition;
b) the access right is a member of the access right set of the prohibition;
c) the policy element lies within either the disjunctive or conjunctive range of the inclusion and exclusion policy element sets of the prohibition, respective of whether the prohibition is a disjunctive or conjunctive variant.

The attribute restriction relation is a ternary relation from UA to AR to PE. It is formally defined as follows:

$UA\_RESTRICT \subseteq UA \times AR \times PE$, where the following properties hold:

$\forall ua \in UA, \forall ar \in AR, \forall pe \in PE$: $((ua, ar, pe) \in UA\_RESTRICT \Leftrightarrow \exists ars \in 2^{AR}, \exists peis \in 2^{PE}, \exists pees \in 2^{PE}$:
$(((ua, ars, peis, pees) \in UA\_DENY\_DISJ \land ar \in ars \land pe \in DisjRange(peis, pees)) \lor$
$((ua, ars, peis, pees) \in UA\_DENY\_CONJ \land ar \in ars \land pe \in ConjRange(peis, pees)))$

4.4.5 Obligation

The obligation relation intuitively is used to screen event notifications of successfully completed access requests and for those events that match a predefined pattern, automatically trigger a predefined response. The user that defines a tuple of the obligation relation must possess adequate authority to carry out the associated response in its entirety at the time the pattern is matched with the event. Otherwise, the response is not attempted.

The pattern and response items of the obligation relation respectively express the conditions of an event pattern and the denotation of the response, as described in NGAC-FA. Processing of the pattern occurs during the matching process, while processing of the response occurs after a match. The execution of an obligation's response occurs immediately after a successful match of an event to the obligation's pattern. That is, the pattern matching and response execution are atomic–no other access request can be processed once a pattern is matched, until the response is complete.

An event pattern and response each represents sentences that must conform to a formal language over its respective alphabet. That is, the syntax of the sequence of symbols for each item must be well formed according to its respective grammar. A pattern and response can each be expressed using different language grammars. Annex A provides an informative example of pattern and response grammars. To interoperate, all NGAC functional entities supporting an application must apply formal languages in a consistent fashion when forming the pattern and the response expressions.

A pattern is a member of the set of all allowable patterns, which is formally defined as follows:

> PATTERN $\subseteq$ $\text{seq}_1\ \Sigma_P$, where Σ_P is the alphabet used to form a logical expression of the conditions to be matched, and each member of PATTERN is well-formed with respect to a specific grammar G_P

Each member of PATTERN shall be represented by a pattern identifier.

A response is a member of the set of all allowable responses, which is defined as follows:

> RESPONSE $\subseteq$ $\text{seq}_1\ \Sigma_R$, where Σ_R is the alphabet used in the denotation of an event response, and each member of RESPONSES is well-formed with respect to a specific grammar G_R

Each member of RESPONSE shall be represented by a response identifier.

The obligation relation is a ternary relation from U to PATTERN to RESPONSE. For each tuple (u, pattern, response) of the obligation relation, u represents the user that established the pattern and response, and under whose authorization the response is carried out. An obligation relation is formally defined as follows:

> OBLIG $\subseteq$ U $\times$ PATTERN $\times$ RESPONSE

4.5 Access authorization

The NGAC authorization decision function controls accesses in terms of processes. The user on whose behalf the process operates must hold sufficient authority over the policy elements involved, in the form of at least one and possibly more privileges. That is, access requests to perform an operation on a policy element, including objects, are issued only from processes acting on behalf of a user, and are granted authorization provided appropriate privileges exist that allow the access, and no restriction exists that prevents the access. If a restriction does exist, the access request is denied.

A tuple of an access request relation AREQ is defined as (p, op, argseq), where $p \in P$, $op \in Op$ and $argseq \in \text{seq}_1$ Arg. The argument sequence, argseq, is a finite sequence of one or more arguments, which is compatible with the scope of the operation. Each argument in the sequence can be one of the following items: a distinct policy element, a set of policy elements, a set of access rights, an event pattern, or a response. That is, an administrative access request comprises an operation and a list of enumerated arguments whose type and order are dictated by the operation.

Access requests involving resource operations affect only objects and are simpler than administrative operations, requiring only a single object attribute in the argument sequence. Access requests that involve administrative operations affect only information persisted in the PIP and are generally more complex, requiring multiple arguments in the argument sequence.

An access request is a ternary relation from P to Op to Argseq. It is formally defined as follows:

> $\text{AREQ} \subseteq \text{P} \times \text{Op} \times \text{Argseq}$, where $\text{Argseq} = \text{seq}_1\ \text{Arg}$ and $\text{Arg} = \{x \mid x \in \text{PE} \lor x \in 2^{\text{PE}} \lor x \in \text{PATTERN} \lor x \in \text{RESPONSE}\}$

For the authorization decision function to determine whether an access request can be granted, it requires a mapping from an operation and argument sequence pair to the set of access rights and policy element pairs the process' user must hold for the request to be granted. The required capabilities mapping is defined as the partial binary function ReqCap from (Op × Argseq) to $2^{(\text{AR}\times\text{PE})}$, such that $\forall op \in \text{Op}, \forall argseq \in \text{Argseq}$: (capset ∈ ReqCap(op, argseq) ⇒ ∀ar ∈ AR, ∀pe ∈ PE: ((ar, pe) ∈ capset, if and only if (ar,pe) is a requisite access right needed to perform the operation op on argseq)).

> $\text{ReqCap} \subseteq (\text{Op}\times\text{Argseq}) \times 2^{(\text{AR}\times\text{PE})}$

The authorization decision function grants a process, p, permission to execute an access request (p, op, pe), provided the following conditions hold for each access right and policy element pair (ar, pe) in one of the capability sets returned by the required capabilities function, ReqCap(op, argseq):

a) there exists a privilege (Process_User(p), ar, pe);
b) there does not exist a process restriction (p, ar, pe) ∈ P_RESTRICT;
c) there does not exist a user restriction (Process_User(p), ar, pe) ∈ U_RESTRICT; and
d) there does not exist a user u that is contained by a user attribute ua, such that u = Process_User(p), and the attribute restriction (ua, ar, pe) ∈ UA_RESTRICT exists.

Otherwise, the requested access is denied.

The authorization decision function is a mapping from domain AREQ to codomain {grant, deny}. It is formally defined as follows:

> Authorization_Decision ⊆ AREQ×{grant, deny}

$$
\begin{array}{l}
\forall p \in \text{P}, \forall op \in \text{Op}, \forall argseq \in \text{Argseq}: ((p, op, argseq) \in \text{AREQ} \Rightarrow \\
(\text{Authorization\_Decision}((p, op, argseq)) = \text{grant} \Leftrightarrow \\
\exists capset \in \text{ReqCap}(op, argseq): \forall ar \in \text{AR}, \forall pe \in \text{PE}: ((ar, pe) \in capset \Rightarrow \\
(\text{Process\_User}(p), ar, pe) \in \text{PRIVILEGE} \land \\
(p, ar, pe) \notin \text{P\_RESTRICT} \land (\text{Process\_User}(p), ar, pe) \notin \text{U\_RESTRICT}) \land \\
\forall ua \in \text{UA}, \nexists u \in \text{Users}(ua): (u = \text{Process\_User}(p) \land (ua, ar, pe) \in \text{UA\_RESTRICT})))); \\
\text{otherwise, Authorization\_Decision}((p, op, argseq)) = \text{deny}
\end{array}
$$

The computation of an authorization decision is illustrated in Figure 1. The thick arrows depict the derivation of the privilege and restriction relations respectively from the association and prohibition relations that reside in the PIP, while the thin arrows depict the use of those derived relations in reaching an authorization decision.

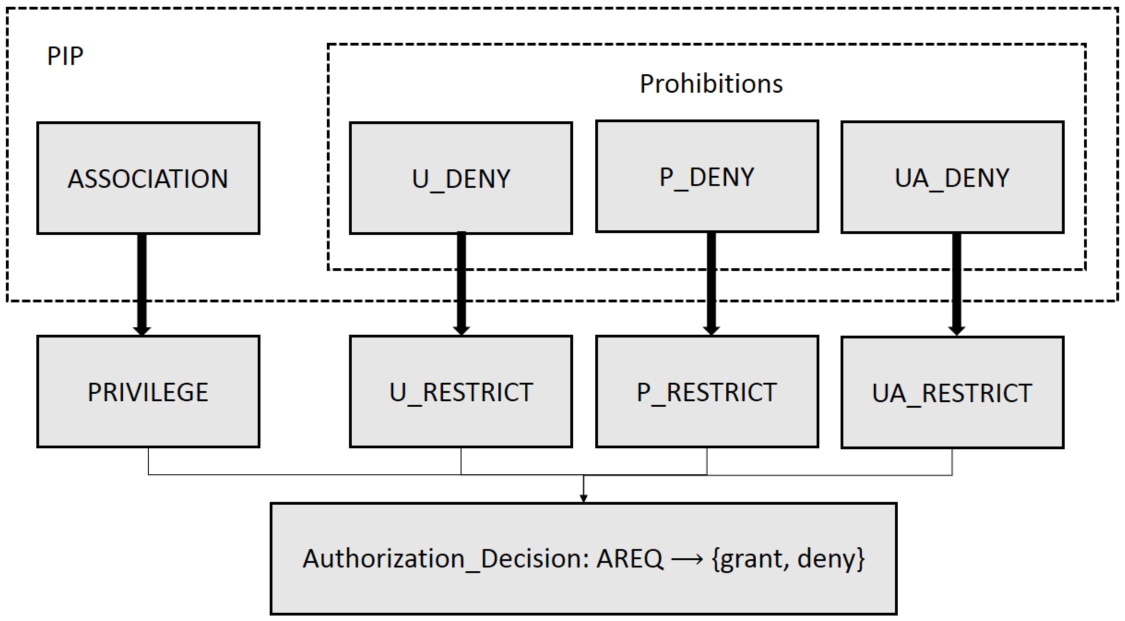

Figure 1: NGAC Authorization Decision Function

5 Administrative Commands

5.1 Overview

The NGAC access control model is in essence a finite state machine. The basic elements and containers, and the relations that define the prevailing access rights between these entities, collectively constitute the authorization state of the policy maintained at the PIP. A change in state or state transition occurs when an access request involving an administrative operation is granted by the authorization decision function and successfully carried out. A change in state may also occur when an obligation is triggered and its response is successfully carried out.

Administrative commands define the behavior of state transitions that occur in the creation, deletion and maintenance of NGAC data elements and relations. An administrative command is performed to carry out a valid administrative access request for a user. An administrative command may also be performed as part of the response for a defined obligation that has been triggered.

Clause 5 defines the semantics for the core NGAC administrative commands. The semantic definitions specify the correct behavior expected of commands, which is necessary to maintain the integrity of the NGAC framework. Access to protected resources via administrative commands can occur only when the access has first been granted by the authorization decision function, thus ensuring that the process attempting access has sufficient authorization to carry out the command.

5.2 Semantic definitions

Each administrative command describes specific changes made to the authorization state maintained by the NGAC. The syntax and notation used to specify semantic behavior should not be interpreted as programming statements and instead, should be interpreted as changes to data structures maintained at the PIP, which occur when a command is correctly invoked. Behavioral aspects other than those that pertain to security are outside the scope of this standard.

Administrative commands exist to:

a) create the following policy elements (see 5.2.1):
   - a user in a user attribute;
   - an object in an object attribute;
   - a policy class;
   - a user attribute in a policy class;
   - a user attribute in a user attribute;
   - an object attribute in a policy class; and
   - an object attribute in an object attribute.

b) delete the following policy elements (see 5.2.2):
   - a user
   - an object;
   - a user attribute;
   - an object attribute; and
   - a policy class.

c) create and delete the following entities (see 5.2.3 and 5.2.4):
   - a resource operation;
   - an administrative operation;
   - an access right;

- a set of access rights;
- a set of policy elements;
- a set of inclusive policy elements;
- a set of exclusive policy elements;
- an event pattern; and
- an event response.

d) create and delete the following relations (see 5.2.5 and 5.2.6):

- assignments;
- associations;
- user-based prohibitions;
- process-based prohibitions;
- attribute-based prohibitions; and
- obligations.

Administrative commands require the orderly build up and tear down of policy representation. Pre-conditions are defined for each administrative command. Pre-conditions denote requirements. They are expressed as a logical expression that must be satisfied for the command to be performed. Predicates appearing on separate lines are conjoined together by default.

The pre-conditions for administrative commands ensure that the arguments supplied as formal parameters are of the correct type, and that the basic properties of the model are observed. Pre-conditions appear before the body of the command. The body is enclosed with a pair of braces and specifies the effect that takes place on the authorization state of the system. By convention, unprimed variables represent state before the command occurs and primed variables represent the change in state due to the command. Although multiple statements may apply to a command, the effect shall be atomic. That is, either all the statements apply, or no change occurs to the authorization state. Comments may appear anywhere in a command. They shall apply to a single line only and begin with double, forward slashes (vis., //).

Some foundational data elements and relations are needed to specify the behavior of administrative commands. All commands depend on a basic data type called ID, which refers to an opaque identifier. An identifier is a finite sequence of bytes, whose characteristics and interpretation are left unspecified. A set called GUID maintains a list of all identifiers allocated to entities of the various classes of elements used to define policy. GUID is used to ensure that an identifier is assigned to only one entity. These data elements are defined formally as follows:

$\text{ID} = \text{seq}_1 \{0000\text{b}, \ldots, 1111\text{b}\}$
$\text{GUID} \subseteq \text{ID}$

Several other data elements and relations are also needed to represent sets whose members represent defined subsets of certain types of entities, namely access rights, exclusive policy elements and inclusive policy elements.

$\text{ARset} \subseteq \text{GUID}$
$\text{ARmap} \subseteq \text{GUID} \times \text{iseq AR}$

$\text{PEIset} \subseteq \text{GUID}$
$\text{PEImap} \subseteq \text{GUID} \times \text{iseq PE}$

$\text{PEEset} \subseteq \text{GUID}$
$\text{PEEmap} \subseteq \text{GUID} \times \text{iseq PE}$

An initial state (i.e., the state immediately after initialization of the NGAC framework) is defined in terms of the abstract data elements and relations and predicates that stipulate the initial conditions of the system. The NGAC authorization state is initialized to zero or empty as described in the administrative command below. The authorization state can then be populated by the security authority with supported elements, such as inherent administrative operations and associated access rights, and tailored to the specifics of the operating environment with other elements, such as resource operations and access rights.

```
InitialState ()
    {
        // initialize policy elements
        U′ = ∅
        O′ = ∅
        UA′ = ∅
        OA′ = ∅
        PC′ = ∅
        P′ = ∅
        PE′ = ∅
        // initialize relations
        ASSIGN′ = ∅
        ASSOCIATION′ = ∅
        U_CONJ_DENY′ = ∅
        U_DISJ_DENY′ = ∅
        P_CONJ_DENY′ = ∅
        P_DISJ_DENY′ = ∅
        UA_CONJ_DENY′ = ∅
        UA_DISJ_DENY′ = ∅
        OBLIG′ = ∅
        // initialize other required entities
        GUID′ = ∅
        Process_User′ = ∅
        AOP′ = ∅
        ROP′ = ∅
        OP′ = ∅
        AR′ = ∅
        ARset′ = ∅
        ARmap′ = ∅
        PEIset′ = ∅
        PEImap′ = ∅
        PEEset′ = ∅
        PEEmap′ = ∅
        PATTERN′ = ∅
        PATTERNmap′ = ∅
        RESPONSE′ = ∅
        RESPONSEmap′ = ∅
    }
```

5.2.1 Element creation

The semantic descriptions of element creation commands describe the state changes that occur with the addition of new policy elements to the policy representation.

```
CreateUinUA (x:ID, y:ID)  // add user x to the policy representation and assign it to user attribute y
    $x \notin U$
    $x \notin GUID$
    $y \in UA$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $U' = U \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }

CreateUAinUA (x:ID, y:ID)  // add user attribute x and assign it to user attribute y
    $x \notin UA$
    $x \notin GUID$
    $y \in UA$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $UA' = UA \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }

CreateUAinPC (x:ID, y:ID)  // add user attribute x and assign it to policy class y
    $x \notin UA$
    $x \notin GUID$
    $y \in PC$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $UA' = UA \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }

CreateOinOA (x:ID, y:ID)  // add object x and assign it to object attribute y
    $x \notin GUID$
    $x \notin O$
    $x \notin OA$
    $y \in OA$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $O' = O \cup \{x\}$
        $OA' = OA \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }

CreateOAinOA (x:ID, y:ID)  // add object attribute x and assign it to object attribute y
    $x \notin OA$
    $x \notin GUID$
    $y \in OA$
    $y \notin O$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $OA' = OA \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }

CreateOAinPC (x:ID, y:ID)  // add object attribute x and assign it to policy class y
    $x \notin OA$
    $x \notin GUID$
    $y \in PC$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $OA' = OA \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }

CreatePC (x:ID)  // add a policy class x to the policy representation
    $x \notin PC$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $PC' = PC \cup \{x\}$
        $PE' = PE \cup \{x\}$
    }
```

5.2.2 Element deletion

The semantic descriptions of element deletion commands describe the state changes that occur with the removal of existing policy elements from the policy representation.

```
DeleteU (x: ID)  // remove user x from the policy representation
    $x \in U$
    $x \in GUID$
    $\nexists p \in P: (p, x) \in Process\_User$  // ensure no processes that operate on behalf of x exist
    $\nexists (a, b) \in ASSIGN: x = a$           // ensure no assignments emanating from the user exist
    // ensure no associations exist in which the user is the third element of the tuple
    $\nexists (a, b, c) \in ASSOCIATION: x = c$
    // ensure no prohibitions exist for the user
    $\nexists (a, b, c, d) \in U\_DENY\_DISJ: a = x$
    $\nexists (a, b, c, d) \in U\_DENY\_CONJ: a = x$
    // ensure no inclusive element sets exist that involve the user
    $\nexists (a, b) \in PEImap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    // ensure no exclusive element sets exist that involve the user
    $\nexists (a, b) \in PEEmap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    // ensure no obligations exist defined by the user
    $\nexists (a, b, c) \in OBLIG: a = x$
    {
        $GUID' = GUID \setminus \{x\}$
        $U' = U \setminus \{x\}$
        PE′ = PE \ {x}
    }

DeleteUA (x: ID)   // remove user attribute x from the policy representation
    x ∈ UA
    x ∈ GUID
    // ensure no assignments emanating from or to the user attribute exist
    ∄(a, b) ∈ ASSIGN: (x = a ⋁ x = b)
    // ensure no associations exist in which the user attribute is the first or last element of the tuple
    ∄(a, b, c) ∈ ASSOCIATION: (x = a ⋁ x = c)
    // ensure no attribute prohibitions exist in which the user attribute is the first element of the tuple
    ∄(a, b, c, d) ∈ UA_DENY_DISJ: x = a
    ∄(a, b, c, d) ∈ UA_DENY_CONJ: x = a
    // ensure no inclusive element sets exist that involve the user attribute
    ∄(a, b) ∈ PEImap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    // ensure no exclusive element sets exist that involve the user attribute
    ∄(a, b) ∈ PEEmap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    {
        GUID′ = GUID \ {x}
        UA′ = UA \ {x}
        PE′ = PE \ {x}
    }

DeleteO (x: ID)  // remove object x from the policy representation
    x ∈ O
    x ∈ OA
    x ∈ GUID
    ∄(a, b) ∈ ASSIGN: x = a     // ensure no assignments emanating from the object exist
    // ensure no associations exist in which the object/object attribute is the third element of the tuple
    ∄(a, b, c) ∈ ASSOCIATION: x = c
    // ensure no inclusive element sets exist that involve the object
    ∄(a, b) ∈ PEImap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    // ensure no exclusive element sets exist that involve the object
    ∄(a, b) ∈ PEEmap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    {
        GUID′ = GUID \ {x}
        O′ = O \ {x}
        OA′ = OA \ {x}
        PE′ = PE \ {x}
    }

DeleteOA (x: ID)   // remove object attribute x from the policy representation
    x ∈ OA
    x ∉ O
    x ∈ GUID
    // ensure no assignments emanating from or to the object attribute exist
    ∄(a, b) ∈ ASSIGN: (x = a ⋁ x = b)
    // ensure no associations exist in which the object attribute is the third element of the tuple
    ∄(a, b, c) ∈ ASSOCIATION: x = c
    // ensure no inclusive element sets exist that involve the object attribute
    ∄(a, b) ∈ PEImap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    // ensure no exclusive element sets exist that involve the object attribute
    ∄(a, b) ∈ PEEmap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    {
        GUID′ = GUID \ {x}
        OA′ = OA \ {x}
        PE′ = PE \ {x}
    }
```

```
DeletePC (x: ID)   // remove policy class x from the policy representation
    x ∈ PC
    x ∈ GUID
    ∄(a, b) ∈ ASSIGN: x = b   // ensure no assignments emanating to the policy class exist
    // ensure no associations exist in which the policy class is the third element of the tuple
    ∄(a, b, c) ∈ ASSOCIATION: x = c
    // ensure no inclusive element sets exist that involve the policy class
    ∄(a, b) ∈ PEImap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    // ensure no exclusive element sets exist that involve the policy class
    ∄(a, b) ∈ PEEmap: ∃i ∈ {1, ..., #b}: (b (i) = x)
    {
        GUID′ = GUID \ {x}
        PC′ = PC \ {x}
        PE′ = PE \ {x}
    }
```

5.2.3 Entity creation

The semantic descriptions of element creation commands describe the state changes that occur with the addition of new entities to the policy representation.

```
CreateP (x:ID, y:ID)  // add process x and map it to user y
    x ∉ P
    x ∉ GUID
    y ∈ U
    y ∈ GUID
    {
        GUID′ = GUID ∪ {x}
        P′ = P ∪ {x}
        Process_User′ = Process_User ∪ {(x, y)}
    }

CreateROp (x: ID)  // add a resource operation to the policy representation
    x ∉ Op
    x ∉ GUID
    {
        GUID′ = GUID ∪ {x}
        Op′ = Op ∪ {x}
        ROp′ = ROp ∪ {x}
    }

CreateAOp (x: ID)  // add an administrative operation to the policy representation
    x ∉ Op
    x ∉ GUID
    {
        GUID′ = GUID ∪ {x}
        Op′ = Op ∪ {x}
        AOp′ = AOp ∪ {x}
    }
```

```
CreateAR (x: ID)  // add an access right to the policy representation
    $x \notin AR$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $AR' = AR \cup \{x\}$
    }

CreateARset (x: ID, y: iseq ID)  // add a set of defined access rights to the policy representation
    $x \notin ARset$
    $x \notin GUID$
    $\forall i \in \{1, ..., \#y\}: (y (i) \in AR)$  // ensure each element of the sequence is an access right
    {
        $GUID' = GUID \cup \{x\}$
        $ARset' = ARset \cup \{x\}$            // maintains set of identifiers for defined access right sets
        $ARmap' = ARmap \cup \{(x, y)\}$   // maintains mapping from ARset identifiers to AR sequences
    }

CreatePEIset (x: ID, y: iseq ID)  // add a set of inclusion policy elements to the representation
    $x \notin PEIset$
    $x \notin GUID$
    $\forall i \in \{1, ..., \#y\}: (y (i) \in PE)$
    {
        $GUID' = GUID \cup \{x\}$
        $PEIset' = PEIset \cup \{x\}$
        $PEImap' = PEImap \cup \{(x, y)\}$
    }

CreatePEEset (x: ID, y: iseq ID)  // add a set of exclusion policy elements to the representation
    $x \notin PEEset$
    $x \notin GUID$
    $\forall i \in \{1, ..., \#y\}: (y (i) \in PE)$
    {
        $GUID' = GUID \cup \{x\}$
        $PEEset' = PEEset \cup \{x\}$
        $PEEmap' = PEEmap \cup \{(x, y)\}$
    }

CreatePattern (x: ID, y: $seq_1\ \Sigma_P$)  // add an event pattern to the policy representation
    $x \notin PATTERN$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $PATTERN' = PATTERN \cup \{x\}$
        $PATTERNmap' = PATTERNmap \cup \{(x, y)\}$
    }

CreateResponse (x: ID, y: $seq_1\ \Sigma_R$)  // add an event response to the policy representation
    $x \notin RESPONSE$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $RESPONSE' = RESPONSE \cup \{x\}$
        $RESPONSEmap' = RESPONSEmap \cup \{(x, y)\}$
    }
```

5.2.4 Entity deletion

The semantic descriptions of element deletion commands describe the state changes that occur with the removal of existing entities from the policy representation.

```
DeleteP (x: ID)   // remove process x from the policy representation
    $x \in P$
    $x \in GUID$
    $\exists_1 u \in U: u = Process\_User(x)$               // ensure the process maps to a user
    $\nexists (a, b, c, d) \in P\_DENY\_CONJ: x = a$  // ensure no outstanding conjunctive prohibitions exist
    $\nexists (a, b, c, d) \in P\_DENY\_DISJ: x = a$   // ensure no outstanding disjunctive prohibitions exist
    {
        $GUID' = GUID \setminus \{x\}$
        $Process\_User' = Process\_User \setminus \{(x, Process\_User(x))\}$
    }

DeleteROp (x: ID)  // remove a resource operation from the policy representation
    $x \in ROp$
    $x \in Op$
    $x \in GUID$
    {
        $GUID' = GUID \setminus \{x\}$
        $Op' = Op \setminus \{x\}$
        $ROp' = ROp \setminus \{x\}$
    }

DeleteAOp (x: ID)  // remove an administrative operation from the policy representation
    $x \in AOp$
    $x \in Op$
    $x \in GUID$
    {
        $GUID' = GUID \setminus \{x\}$
        $Op' = Op \setminus \{x\}$
        $AOp' = AOp \setminus \{x\}$
    }

DeleteAR (x: ID)  // remove an access right from the policy representation
    $x \in AR$
    $x \in GUID$
    // ensure the access right does not belong to any access rights set
    $\nexists (a, b) \in ARmap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    {
        $GUID' = GUID \setminus \{x\}$
        $AR' = AR \setminus \{x\}$
    }

DeleteARset (x: ID)  // remove an access rights set from the policy representation
    $x \in ARset$
    $x \in GUID$
    $\exists_1 (a, b) \in ARmap: x = a$
    // ensure no associations exist in which the access right set is the second element of the tuple
    $\nexists$(a, b, c) $\in$ ASSOCIATION: x = b
    // ensure no disjunctive user prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ U_DENY_DISJ: x = b
    // ensure no conjunctive user prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ U_DENY_CON: x = b
    // ensure no disjunctive process prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ P_DENY_DISJ: x = b
    // ensure no conjunctive process prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ P_DENY_CON: x = b
    // ensure no disjunctive attribute prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ UA_DENY_DISJ: x = b
    // ensure no conjunctive attribute prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ UA_DENY_CON: x = b
    {
        GUID′ = GUID \ {x}
        ARset′ = ARset \ {x}
        ARmap′ = ARmap \ {(x, y)}  // the associated sequence of access rights is removed
    }

DeletePEIset (x: ID)  // remove a set of inclusion policy elements from the representation
    x $\in$ PEIset
    x $\in$ GUID
    $\exists_1$(a, b) $\in$ PEImap: x = a
    // ensure no disjunctive user prohibitions exist that involve the policy element set
    $\nexists$(a, b, c, d) $\in$ U_DENY_DISJ: x = c
    // ensure no conjunctive user prohibitions exist that involve the policy element set
    $\nexists$(a, b, c, d) $\in$ U_DENY_CON: x = c
    // ensure no disjunctive process prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ P_DENY_DISJ: x = c
    // ensure no conjunctive process prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ P_DENY_CON: x = c
    // ensure no disjunctive attribute prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ UA_DENY_DISJ: x = c
    // ensure no conjunctive attribute prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ UA_DENY_CON: x = c
    {
        GUID′ = GUID \ {x}
        PEIset′ = PEIset \ {x}
        PEImap′ = PEImap \ {(x, y)}  // the associated sequence of policy elements is removed
    }

DeletePEEset (x: ID)  // remove a set of exclusion policy elements from the representation
    x $\in$ PEEset
    x $\in$ GUID
    $\exists_1$(a, b) $\in$ PEEmap: x = a
    // ensure no disjunctive user prohibitions exist that involve the policy element set
    $\nexists$(a, b, c, d) $\in$ U_DENY_DISJ: x = d
    // ensure no conjunctive user prohibitions exist that involve the policy element set
    $\nexists$(a, b, c, d) $\in$ U_DENY_CON: x = d
    // ensure no disjunctive process prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ P_DENY_DISJ: x = d
    // ensure no conjunctive process prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ P_DENY_CON: x = c
    // ensure no disjunctive attribute prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ UA_DENY_DISJ: x = c
    // ensure no conjunctive attribute prohibitions exist that involve the access right set
    $\nexists$(a, b, c, d) $\in$ UA_DENY_CON: x = c
    {
        GUID′ = GUID \ {x}
        PEEset′ = PEEset \ {x}
        PEEmap′ = PEEmap \ {(x, y)}  // the associated sequence of policy elements is removed
    }
```

```
DeletePattern (x: ID)  // from an event pattern from the policy representation
    x $\in$ PATTERN
    x $\in$ GUID
    // ensure no obligations exist that involve the pattern
    $\nexists$(a, b, c) $\in$ OBLIG: x = b
    {
        GUID′ = GUID \ {x}
        PATTERN′ = PATTERN \ {x}
        PATTERNmap′ = PATTERNmap \ {(x, y)}
    }
```

```
DeleteResponse (x: ID)  // delete an event response from the policy representation
    x $\in$ RESPONSE
    x $\in$ GUID
    // ensure no obligations exist that involve the response
    $\nexists$(a, b, c) $\in$ OBLIG: x = c
    {
        GUID′ = GUID \ {x}
        RESPONSE′ = RESPONSE \ {x}
        RESPONSEmap′ = RESPONSEmap  \ {(x, y)}
    }
```

5.2.5 Relation formation

The semantic descriptions of relation formation commands describe state changes that occur with the addition of tuples to existing relations and functions in the policy representation.

```
CreateAssign (x:ID, y:ID)  // add tuple (x, y) to the assignment relation
    x $\in$ PE
    y $\in$ PE
    ((x $\in$ U $\wedge$ y $\in$ UA)  $\vee$  (x $\in$ UA $\wedge$ y $\in$ UA)  $\vee$  (x $\in$ UA $\wedge$ y $\in$ PC)  $\vee$
              (x $\in$ OA $\wedge$ y $\in$ (OA \ O))  $\vee$  (x $\in$ (OA \ O) $\wedge$ y $\in$ PC))
    x $\neq$ y  // prevents the creating of a loop
    (x, y) $\notin$ ASSIGN
    // ensure that no chain of assignments will result, which creates a cycle (i.e., if x and y are both
    // members of UA or OA, then there cannot already exist a series of assignments from y to x)
    (x, y $\in$ UA $\vee$ x, y $\in$ OA)  $\Rightarrow$  $\nexists$s $\in$ iseq$_1$ PE: (#s > 1 $\wedge$ $\forall$i $\in$ {1,...,(#s - 1)}:
    ((s (i), s (i+1)) $\in$ ASSIGN) $\wedge$ (s (1) = y  $\wedge$  s (#s) = x)
    {
        ASSIGN′ = ASSIGN $\cup$ {(x, y)}
    }
```

```
CreateAssoc (x:ID, y:ID, z:ID)  // add tuple (x, y, z) to the association relation
    x $\in$ UA
    y $\in$ ARset
```

```
        $z \in PE$
        $(x, y, z) \notin ASSOCIATION$
        // ensure no duplicate association exists
        $\nexists(a, b, c) \in ASSOCIATION: (a = x \wedge ran\ ARmap(b) = ran\ ARmap(y) \wedge c = z)$
        {
            $ASSOCIATION' = ASSOCIATION \cup \{(x, y, z)\}$
        }

    CreateConjUserProhibit (w: ID, x:ID, y:ID, z:ID)  // add tuple (w, x, y, z) to the prohibition relation
        $w \in U$
        $x \in ARset$
        $y \in PEIset$
        $z \in PEEset$
        $(w, x, y, z) \notin U\_DENY\_CONJ$
        // ensure no duplicate prohibition exists
        $\nexists(a, b, c, d) \in U\_DENY\_CONJ: (a = w \wedge ran\ ARmap(b) = ran\ ARmap(x) \wedge$
        $ran\ PEImap(c) = ran\ PEImap(y) \wedge ran\ PEEmap(d) = ran\ PEEmap(z))$
        {
            $U\_DENY\_CONJ' = U\_DENY\_CONJ \cup \{(w, x, y, z)\}$
        }

    CreateConjProcessProhibit (w: ID, x:ID, y:ID, z:ID)  // add tuple (w, x, y, z) to the prohibition relation
        $w \in P$
        $x \in ARset$
        $y \in PEIset$
        $z \in PEEset$
        $(w, x, y, z) \notin P\_DENY\_CONJ$
        // ensure no duplicate prohibition exists
        $\nexists(a, b, c, d) \in P\_DENY\_CONJ: (a = w \wedge ran\ ARmap(b) = ran\ ARmap(x) \wedge$
        $ran\ PEImap(c) = ran\ PEImap(y) \wedge ran\ PEEmap(d) = ran\ PEEmap(z))$
        {
            $P\_DENY\_CONJ' = P\_DENY\_CONJ \cup \{(w, x, y, z)\}$
        }

    CreateConjAttributeProhibit (w: ID, x:ID, y:ID, z:ID)  // add tuple (w, x, y, z) to the prohibition relation
        $w \in UA$
        $x \in ARset$
        $y \in PEIset$
        $z \in PEEset$
        $(w, x, y, z) \notin UA\_DENY\_CONJ$
        // ensure no duplicate prohibition exists
        $\nexists(a, b, c, d) \in UA\_DENY\_CONJ: (a = w \wedge ran\ ARmap(b) = ran\ ARmap(x) \wedge$
        $ran\ PEImap(c) = ran\ PEImap(y) \wedge ran\ PEEmap(d) = ran\ PEEmap(z))$
        {
            $UA\_DENY\_CONJ' = UA\_DENY\_CONJ \cup \{(w, x, y, z)\}$
        }
```

The disjunctive forms of user, process and attribute-based prohibition formation are defined similarly to their conjunctive counterparts above.

```
    CreateOblig (x:ID, y:ID, z:ID)  // add tuple (x, y, z) to the obligation relation
        $x \in U$
        $y \in PATTERN$
        $z \in RESPONSE$
        $(x, y, z) \notin OBLIG$
        // ensure no duplicate (i.e., identical sentences, not semantic equivalents) obligation exists
        $\nexists(a, b, c) \in OBLIG: (a = x \ \wedge \ \#PATTERNmap(b) = \#PATTERNmap(y) \ \wedge$
        $\forall i \in \{1, ..., \#PATTERNmap(b)\}: PATTERNmap(b)(i) = PATTERNmap(y)(i) \ \wedge$
        $\#RESPONSEmap(c) = \#RESPONSEmap(z) \ \wedge$
        $\forall i \in \{1, ..., \#RESPONSEmap(c)\}: RESPONSEmap(c)(i) = RESPONSEmap(z)(i))$
        {
            $OBLIG' = OBLIG \cup \{(x, y, z)\}$
        }

    CreatePUmapping (x:ID, y:ID)
        $x \in P$
        $y \in U$
        $(x, y) \notin Process\_User$
        $\nexists z \in P: (x, z) \in Process\_User$   // ensure no other user already assigned to the process
        {
            $Process\_User' = Process\_User \cup \{(x, y)\}$
        }
```

5.2.6 Relation rescindment

The semantic descriptions of relation rescindment commands describe state changes that occur with the removal of tuples from existing relations and functions in the policy representation.

```
    DeleteAssign (x:ID, y:ID)  // remove tuple (x, y) from the assignment relation
        $((x \in U \wedge y \in UA) \ \vee \ (x \in UA \wedge y \in UA) \ \vee \ (x \in UA \wedge y \in PC) \ \vee$
                    $(x \in OA \wedge y \in (OA \backslash O)) \ \vee \ (x \in (OA \backslash O) \wedge y \in PC))$
        $(x, y) \in ASSIGN$
        // ensure that if no other assignment emanates from x, no assignments emanate to x
        $\nexists z \in PE: (x, z) \in ASSIGN \ \Rightarrow \ \nexists v \in PE: (v, x) \in ASSIGN$
        {
            $ASSIGN' = ASSIGN \backslash \{(x, y)\}$
        }

    DeleteAssoc(x:ID, y:ID, z:ID)  // remove tuple (x, y, z) from the association relation
        $x \in UA$
        $y \in ARset$
        $z \in PE$
        $(x, y, z) \in ASSOCIATION$
        {
            $ASSOCIATION' = ASSOCIATION \backslash \{(x, y, z)\}$
        }

    DeleteConjUserProhibit (w: ID, x:ID, y:ID, z:ID)  // remove tuple from user prohibition relation
        $w \in U$
        $x \in ARset$
        $y \in PEIset$
        $z \in PEEset$
        $(w, x, y, z) \in U\_DENY\_CONJ$
        {
            $U\_DENY\_CONJ' = U\_DENY\_CONJ \backslash \{(w, x, y, z)\}$
        }
```

```
DeleteConjProcessProhibit (w: ID, x:ID, y:ID, z:ID)  // remove tuple from process prohibition relation
    $w \in P$
    $x \in ARset$
    $y \in PEIset$
    $z \in PEEset$
    $(w, x, y, z) \in P\_DENY\_CONJ$
    {
        $P\_DENY\_CONJ' = P\_DENY\_CONJ \setminus \{(w, x, y, z)\}$
    }

DeleteConjAttributeProhibit (w: ID, x:ID, y:ID, z:ID)  // remove tuple from process prohibition relation
    $w \in UA$
    $x \in ARset$
    $y \in PEIset$
    $z \in PEEset$
    $(w, x, y, z) \in UA\_DENY\_CONJ$
    {
        $UA\_DENY\_CONJ' = UA\_DENY\_CONJ \setminus \{(w, x, y, z)\}$
    }
```

The disjunctive forms of user, process and attribute-based prohibition rescindment are defined similarly to their conjunctive counterparts above.

```
DeleteOblig (x:ID, y: ID, z: ID)  // remove tuple (x, y, z) from the obligation relation
    $x \in U$
    $y \in PATTERN$
    $z \in RESPONSE$
    $(x, y, z) \in OBLIG$
    {
        $OBLIG' = OBLIG \setminus \{(x, y, z)\}$
    }

DeletePUmapping (x: ID, y: ID)
    $x \in P$
    $y \in U$
    $(x, y) \in Process\_User$
    {
        $Process\_User' = Process\_User \setminus \{(x, y)\}$
    }
```

Annex A
(Informative)
Pattern and Response Grammars

A.1 Overview

Obligations represent potential changes to the authorization state of the policy. They are used when one of more administrative actions need to be carried out under a specific, predefined set of circumstances. Insofar as their ability to change the authorization state is concerned, obligations have a similar effect as administrative commands. Important differences exist between them, however. Administrative commands are typically carried out in response to an access request that has first been subjected to the authorization decision function for approval, which ensures that the requestor holds sufficient authorization. When the circumstances of a defined obligation are matched, the actions that make up the associated response are carried out automatically, under the authorization held by the user that defined the obligation. The triggering of the obligation allows resolution of any unresolved terms appearing in the response with the details of the triggering event, and also verification that the user that defined the obligation holds sufficient authorization to carry out the response.

The benefits of obligations are twofold: complex policies that require more dynamic treatment than the GOADS policy representation provides can be accommodated, and administrative actions that require repetitive performance of administrative commands can be automated. The main drawback is the potential to cause grave harm to the authorization state through error or intent. The former can be mitigated by thoroughly testing any obligation before it is enabled, and the latter by taking judicious care when employing obligations, preferably granting only trusted individuals the authorization to define obligations and restricting the scope of those individual's authority to well-defined groups of policy elements.

As described in clause 4, each defined obligation is represented by a triple of the OBLIG relation of the form (user, pattern, response). From the perspective of the formal model, the pattern and response terms are simply a sequence of symbols from some alphabet. Therefore, the recognition and treatment of the acceptable symbol sequences that are possible for each item must be handled outside of the model using a different formalism well suited for this goal.

A grammar is a formal mechanism that can be used either to enumerate the sentences of a language or to determine whether a given sentence (i.e., a sequence of symbols from some alphabet) belongs to the language described by the grammar. The grammars for the pattern and response terms of an obligation relation can be specified using Backus-Naur Form (BNF) notation. A BNF grammar comprises a set of symbols and production rules, which formally describe a language. The pattern and response grammars, therefore, each defines a formal language whose respective sentences are conveyed by the corresponding terms of an obligation.

The following conventions are followed in the BNF notation used in the specifications below. Non-terminal symbols are enclosed by left and right-pointing angle braces (vis., ⟨⟩). Terminal symbols are bolded. Undefined symbols that lie outside the grammar (e.g., function calls) are italicized. Procedures and functions are specified in the syntax of the "C" programming language. Production rules use a special symbol (vis., ::=) as a replacement operator that separates a non-terminal symbol on the left-hand side from the replacement rewrite expression on the right-hand side. A vertical bar (vis., **|**) separates alternative rewrite expressions. Concatenation of adjacent symbols is implicit and takes precedence over **|**, which takes precedence over ::=. Paired square brackets group together expressions that are optional, and paired curly brackets group together expressions that may occur zero or more times. These grouping designators, which are drawn from the Extended BNF notation, take precedence over other operators.

A.2 Event pattern grammar

A.2.1 Base specification

The event pattern grammar specifies an event, such as a specific operation performed on an object by a process running on behalf of a user that has certain attributes in some policy classes. The event pattern comprises four components, namely a user specification (see A.2.2), a policy class specification (see A.2.3), an operation specification (see A.2.4), and a policy element specification (see A.2.5). With the exception of the operation specification, all components are optional. In order to apply an event-response rule, the event being processed (the current event) must match every component specification present in the rule's event pattern.

⟨event pattern⟩ ::= [⟨user spec⟩] [⟨pc spec⟩] ⟨op spec⟩ [⟨pe spec⟩]

A.2.2 User specification

If the user specification is present, it denotes the processes of a specific user, of any user, or of any user from a set of users and/or user attributes, or a specific process specified through its identifier. If the user specification is omitted, then by default any event matches this component of the pattern.

⟨user spec⟩ ::= ⟨user⟩ **|** ⟨any user⟩ **|** ⟨process⟩

⟨user⟩ ::= [**user**] *user_name*

⟨any user⟩ ::= **any** [**user**] [**of** ⟨user or attr set⟩]

⟨user or attr set⟩ ::= ⟨user or attr⟩ {**,** ⟨user or attr⟩}

⟨user or attr⟩ ::= ⟨user⟩ **|** ⟨uattr⟩

⟨uattr⟩ ::= **attribute** *attribute_name*

⟨process⟩ ::= **process** *process_id*

A.2.3 Policy class specification

The policy class specification, if present, can specify a particular policy class by name, any policy class, any policy class from a set, all policy classes, or all policy classes from a set. The current event matches the policy class specification if the user is contained in the designated policy classes. If the policy class is omitted, any event matches this component of the pattern.

⟨pc spec⟩ ::= **in** ⟨pc subspec⟩

⟨pc subspec⟩ ::= ⟨pc⟩ **|** ⟨any pc⟩ **|** ⟨each pc⟩

⟨pc⟩ ::= [**policy**] *pc_name*

⟨any pc⟩ ::= **any** [**policy**] [**of** ⟨pc set⟩]

⟨each pc⟩ ::= **each** [**policy**] [**of** ⟨pc set⟩]

⟨pc set⟩ ::= ⟨pc⟩ {**,** ⟨pc⟩}

A.2.4 Operation specification

The mandatory operation specification specifies the event operation by its name, or as any operation, or as any operation from a set of operations.

⟨op spec⟩ ::= **performs** ⟨op subspec⟩

⟨op subspec⟩ ::= ⟨op⟩ **|** ⟨any op⟩

⟨op⟩ ::= [**operation**] *op_name*

⟨any op⟩ ::= **any** [**operation**] [**of** ⟨op set⟩]

⟨op set⟩ ::= ⟨op⟩ {**,** ⟨op⟩}

A.2.5 Policy element specification

If the policy element specification is present, it can specify a policy element by its name, any policy element, any policy elements contained in other policy elements, or any policy element from a set of enumerated policy elements.

⟨pe spec⟩ ::= **on** ⟨pe subspec⟩ [⟨membership subspec⟩]

⟨pe subspec⟩ ::= ⟨pe⟩ **|** ⟨any pe⟩

⟨pe⟩ ::= [**policy element**] *pe_name*

⟨any pe⟩ ::= **any** [**policy element**]

⟨membership subspec⟩ ::= **in** ⟨pe⟩ **|** **of** ⟨pe set⟩

⟨pe set⟩ ::= ⟨pe⟩ {**,** ⟨pe⟩}

A.3 Event response grammar

A.3.1 Base Specification

The response grammar specifies a sequence of conditional actions to be performed by the EPP whenever an event occurs which matches the corresponding event pattern. Actions are prefixed with an optional condition that can be used to specify the existence or nonexistence of some policy element. The types of actions that are defined for a response are create (see A.3.2), assign (see A.3.3), grant (see A.3.4), deny (see A.3.5), and delete (see A.3.6) actions.

⟨response⟩ ::= ⟨conditional action⟩ {**,** ⟨conditional action⟩}

⟨conditional action⟩ ::= [**if** ⟨condition⟩ **then**] ⟨action⟩ {**,** ⟨action⟩}

⟨condition⟩ ::= ⟨factor⟩ {**and** ⟨factor⟩}

⟨factor⟩ ::= [**not**] ⟨cond entity⟩ ⟨rest factor⟩

⟨rest factor⟩ ::= **exists** **|** **in** ⟨cond entity⟩

⟨cond entity⟩ ::= **user** [**attribute**] ⟨name or function call⟩ |
object [**attribute**] ⟨name or function call⟩ |
policy ⟨name or function call⟩

⟨name or function call⟩ ::= *name* **|** *fn_name* **(** [⟨arg part⟩] **)**

⟨arg part⟩ ::= ⟨name or function call⟩ {**,** ⟨name or function call⟩}

⟨action⟩ ::= ⟨create action⟩ |
⟨assign action⟩ |
⟨grant action⟩ |
⟨deny action⟩ |
⟨delete action⟩

A.3.2 Create action specification

A create action creates a policy element and optionally assigns it to another element. The action is introduced by the keyword **create** and specifies the entity to be created, the entity that is to be represented by it, and the containers within which the new entity is to be created and assigned. The entity to create (i.e., ⟨create what⟩) can be either a user, a user attribute, an object, an object attribute, or a policy class. The containers where the new entity is to be created is specified by the ⟨create where⟩ production rule, which can designate the base node of the framework, a policy class, user attribute, or an object attribute. A container name can be specified explicitly or as the result of a function call. Functions are evaluated at run time, as the response is being carried out.

⟨create action⟩ ::= **create** ⟨create what⟩ ⟨create where⟩

⟨create what⟩ ::= ⟨user or obj prefix⟩ [**attribute**] ⟨name or function call⟩ |
policy ⟨name or function call⟩

⟨user or obj prefix⟩ ::= **user | object**

⟨create where⟩ ::= **in** ⟨container⟩

⟨container⟩ ::= ⟨base container⟩ **|** ⟨policy container⟩ **|** ⟨attr container⟩

⟨base container⟩ ::= **base**

⟨policy container⟩ ::= **policy** ⟨name or function call⟩

⟨attr container⟩ ::= ⟨attr prefix⟩ ⟨name or function call⟩

⟨attr prefix⟩ ::= **user attribute | object attribute**

A.3.3 Assign action specification

An assign action creates an assignment, provided that no duplicate assignment already exists. It is introduced by the keyword **assign** followed by two components: the source entity and the destination entities. The entity to be assigned can be a user, a user attribute, an object, or an object attribute. The destination entities can be either a set of containers to which the source entity must be assigned, or (by using the keyword **like**) a user, a user attribute, an object, or an object attribute from which the assignments can be copied.

⟨assign action⟩ ::= **assign** ⟨assign what⟩ [⟨assign to⟩]

⟨assign what⟩ ::= ⟨user or obj prefix⟩ [**attribute**] ⟨name or function call⟩

⟨assign to⟩ ::= **to** ⟨container set⟩ **|** **like** ⟨model entity⟩

⟨container set⟩ ::= ⟨container⟩ {**,** ⟨container⟩}

⟨model entity⟩ ::= ⟨user or obj prefix⟩ [**attribute**] ⟨name or function call⟩

A.3.4 Grant action specification

A grant action creates an association, provided that no duplicate association already exists. The action is introduced by the keyword **grant** and comprises three components: the user attributes, the granted access rights, and the policy elements on which the access rights are granted. The ⟨grant to⟩ component specifies user attributes that receive the granted access rights. A user attribute can be specified explicitly or as the result of a function call. The ⟨grant what⟩ component specifies the access rights granted. The optional ⟨grant on⟩ component, if specified, identifies policy elements targeted by the grant. If omitted, the policy element specified in the optional ⟨policy element⟩ component of the event pattern is targeted, and if that component is also not present, the policy element targeted in the matched event is the one targeted by the grant.

⟨grant action⟩ ::= **grant** ⟨grant to⟩ ⟨grant what⟩ [⟨grant on⟩]

⟨grant to⟩ ::= ⟨uattr spec⟩ {**,** ⟨uattr spec⟩}

⟨uattr spec⟩ ::= [[**user**] **attribute**] ⟨name or function call⟩

⟨grant what⟩ ::= ⟨ar prefix⟩ ⟨granted ar set⟩

⟨ar prefix⟩ ::= **access right | access rights**

⟨granted ar set⟩ ::= *ar_name* {**,** *ar_name*}

⟨grant on⟩ ::= **on** [**policy element**] ⟨name or function call⟩

A.3.5 Deny action specification

A deny action creates a prohibition, provided that no duplicate prohibition already exists. It is introduced by the keyword **deny** and comprises three components. The ⟨deny to⟩ component specifies the user, the user attribute, session, or process that is denied the operations specified by ⟨deny what⟩. If the operand is a user, then the deny operation is user-based. If the operand is a user attribute, then the deny operation is attribute-based. If the operand is a process, then the deny operation is process-based. The ⟨deny what⟩ clause specifies the access rights denied. The ⟨deny on⟩ component specifies a list of referent policy elements to which the denied access rights apply. The list can be prefixed by the keyword **complement**, meaning the complement of the set of policy elements represented by the members of the list. The list is normally interpreted as the union of its members. If prefixed by the keyword **intersection of**, then it is interpreted as the intersection of its members. A list member can also be prefixed by the keyword **complement**, meaning the complement of the set of policy elements represented by that member as a referent container.

⟨deny action⟩ ::= **deny** ⟨deny to⟩ ⟨deny what⟩ ⟨deny on⟩

⟨deny to⟩ ::= **user** [**attribute**] ⟨name or function call⟩ |
process ⟨name or function call⟩

⟨deny what⟩ ::= ⟨ar prefix⟩ ⟨denied ar set⟩

⟨ar prefix⟩ ::= **access right | access rights**

⟨denied ar set⟩ ::= *ar_name* {**,** *ar_name*}

⟨deny on⟩ ::= [**on elements of** [**complement of**] [**intersection of**]] ⟨policy element set⟩

⟨policy element set⟩ ::= ⟨pe container⟩ {**,** ⟨pe container⟩}

⟨pe container⟩ ::= [**complement of**] [**policy element**] ⟨name or function call⟩

A.3.6 Delete action specification

A delete action is introduced by the keyword **delete**. To date, the following subtypes of delete actions are defined: delete assignment relations, delete deny relations, delete grant relations, and created policy elements.

⟨delete action⟩ ::= **delete** ⟨delete subaction⟩

⟨delete subaction⟩ ::= ⟨delete assign subaction⟩ |
⟨delete deny subaction⟩ |
⟨delete grant subaction⟩ |
⟨delete create subaction⟩

⟨delete assign subaction⟩ ::= **assign** ⟨assign what⟩ **to** ⟨container set⟩

⟨delete deny subaction⟩ ::= **deny** ⟨deny to⟩ ⟨deny what⟩ ⟨deny on⟩

⟨delete grant subaction⟩ ::= **grant** ⟨grant to⟩ ⟨grant what⟩ ⟨grant on⟩

⟨delete create subaction⟩ ::= **create** ⟨create what⟩

A.4 Grammar considerations

The pattern and response grammars, or grammar pairs, specified above provide an example of the type of facility that can be provided for expressing policy through obligations. The example grammar pairs are fairly extensive and, because of the English-like sentence syntax, have a high usability factor, which enables users to easily perform administrative command-like manipulation of policy. However, the process used to analyze and interpret sentences, turning them into actions on the PIP, is also extensive and, as a result, complex. For many policies, the available features provide by the example grammars are superfluous and will go unused. Nevertheless, they provide a picture of the type of policy expressions that are conceivable.

Consider, for instance, a user x applying the example grammars to specify the following verbose and terse forms of a simple obligation:

(x, **any user performs operation** *op_name* **on any policy element of** *pe_name*,
deny process *fn_name*() **access right** *ar_name* **on elements of policy element** *name*)

(x, *op_name* **on any of** *pe_name*,
deny process *fn_name*() **access right** *ar_name* **on elements of** *name*)

This sort of obligation is often used in a multi-level security policy to prevent a process (i.e., returned by *fn_name()*) from accessing certain classes of elements, (i.e., those contained by policy element *name*) by nullifying the processes access authority (i.e., represented by *ar_name*), whenever a certain operation (i.e., indicated by *op_name*) is carried out by it (i.e., the process acting on behalf of some user) on certain classes of elements (i.e., those contained by policy element *pe_name*).

This and other possible obligations that are able to be generated using the grammars specified above can also be expressed using other pattern and response grammars. For illustration purposes, the same obligation is expressed below using two different pairs of grammars. The first takes a very simple approach using procedure calls (i.e., if-op-on-container-members and deny-process-access-to-container-members) to represent the pattern and the response terms of the obligation. The second takes a more mathematical approach over the data elements and relations of the model and the properties of an event notification (i.e., denoted by **event.**property) to represent the pattern and the response terms, using predicate calculus notation and administrative command invocations respectively.

(x, if-op-on-container-members(*op_name, pe_name*),
deny-process-access-to-container-members(*ar_name*, *name*)

(x, **event.**operation = *op_name* $\wedge$ (**event.**target, *pe_name*) $\in$ ASSIGN$^+$,
CreateConjProcessProhibit (**event**.process, {*ar_name*}, {*name*}, $\emptyset$))

While each grammar pair can express the same obligation, different pros and cons apply with respect to their implementation and usage and also that of the grammar pair specified earlier. For example, predicate calculus affords the most expressive way to state patterns at a level of detail in which the semantics of the expressions are obvious, while a simple procedure call obscures those details, but requires a less complicated grammar. Similarly, expressing a response as a procedure call allows reuse of the grammar for patterns, but requires programming and thorough testing of the procedure, while invocating an administrative command takes advantage of an existing predefined procedure that has been thoroughly vetted. These sorts of differences are what ultimately determines the choice of grammars to implement and use for expressing obligations.

For a user to create an obligation, it must hold sufficient authorization to carry out the operation on the policy elements that appear in the pattern and response sentences (e.g., *pe_name* and *name* in the above examples). Specifically, the user must hold "obligate" access rights over all identifiable policy elements present in the pattern and response portions of the obligation. Such authority is granted the user through associations and ultimately constrains what policy element references may be appear in the pattern and response sentences of an obligation, as verified during initial parsing of the sentences. This allows the scope of a defined obligation to be constrained to prescribed portions (i.e., subgraphs) of a policy element diagram.

At creation time, an obligation is not fully formed, however, since parts of the obligation cannot be resolved until the event variables pertaining to the access request that triggered it (e.g., the user whose process triggered the obligation and the object that was accessed) are known. It is only at run time when an obligation is matched, that event variables in the response can be resolved with the details of the triggering event. Therefore, to verify that sufficient authorization is in place to carry out the response at run time, the response must be conducted under the auspices of the user that created it, not the user whose actions triggered it. It is essential to perform the verification steps outlined at both creation time and run time to ensure that the security policy is asserted correctly for obligations.

Annex B
(Informative)
Mappings of Existing Access Control Schemes

B.1 Overview

Over the last several decades, numerous policies and access control models have been proposed to address real world security problems. Only a small subset of these policies are able to be enforce through commercially available access control mechanisms, and even a smaller subset can be enforced by any one mechanism. NGAC comprises a functional architecture and a set of relations and data elements, which allow a number of different access control schemes to be implemented using a common set of services. This annex looks at two common access control models, Chinese Wall and Role-Based Access Control (RBAC), and describes how each can be expressed in terms of NGAC data elements and relations. Such a description entails the formulation of a mapping or algorithm for transforming the features and abstractions of the NGAC framework to a given access control model, such that an authorization decision rendered by the NGAC framework would be the same decision as that rendered by the access control model.

Several different mappings may exist between the NGAC model abstractions and those of another access control scheme, each mapping with its own benefits and drawbacks. The main objective of this annex is to demonstrate that at least one mapping exists in which the NGAC abstractions can be shown to capture the capabilities of the other access control model. That is, the mappings described here are intended mainly to indicate the capability of NGAC to capture the functionality of another access control model. They do not imply that a policy supported by another access control model would necessarily be represented in NGAC using the described mapping.

B.2 Chinese wall

B.2.1 Background

The Chinese Wall policy evolved to address conflict-of-interest issues related to consulting activities within banking and other financial disciplines. It provides a good example of dynamic separation of duty constraints present in real-world situations. The stated objective of the Chinese Wall policy and its associated model is to prevent illicit flows of information that can result in conflicts of interest [BREW89]. The Chinese Wall model is based on several key entities: subjects, objects, and security labels. A security label designates the conflict-of-interest class and the company dataset of each object.

The Chinese Wall policy is application-specific in that it applies to a narrow set of activities that are tied to specific business transactions. Consultants or advisors are given access to proprietary information to provide a service for their clients. When a consultant gains access to the competitive practices of two banks, for instance, the consultant essentially obtains insider information that could be used to profit personally or to undermine the competitive advantage of one or both of the institutions.

The Chinese Wall model establishes a set of access rules that forms a firewall or barrier, which prevents a subject from accessing objects on the wrong side of the barrier. It relies on the consultant's data store to be logically organized such that each company dataset belongs to exactly one conflict-of-interest class, and each object belongs to exactly one company dataset or the dataset of sanitized objects within a specially designated, non-conflict-of-interest class. A subject can have access to at most one company dataset in each conflict-of-interest class. However, the choice of dataset is at the subject's discretion. Once a subject accesses (i.e., reads or writes) an object in a company dataset, the only other objects accessible by that subject must lie within the same dataset or within the datasets of a different conflict-of-interest class. In addition, a subject can write to a dataset only if it does not have read access to an object containing unsanitized information, which resides in a company dataset different from the one for which write access is requested.

Limitations in the formulation of the Chinese Wall model have been noted previously. For example, once a subject has read objects from two or more company datasets, it can no longer write to any, and once a subject has read objects from exactly one company dataset, it can write only to that dataset. Moreover, the policy rules of the model are more restrictive than necessary to meet the stated conflict-of-interest avoidance objective. For instance, as already mentioned, once a subject has read objects from two or more company datasets, it can no longer write to any data set. However, if the datasets were in different conflict-of-interest classes, no violation of the policy would result were the subject allowed to write to those objects. That is, while the policy rules are sufficient to preclude a conflict of interest from occurring, they are not necessary from a formal logic perspective, since actions that do not incur a conflict of interest are also prohibited by the rules.

B.2.2 Mapping considerations

The Chinese Wall policy and model as originally defined by Brewer and Nash is used to formulate a mapping from GOADS [BREW89]. However, the model is interpreted slightly differently to resolve one of the limitations noted above. Instead of treating both users and programs acting on a user's behalf collectively as subjects, they are differentiated from one another, such that the stated policy continues to apply to programs, but not to users, who are trusted as individuals to honor Chinese Wall barriers [SAND92].

The following GOADS policy elements and relations can be used to represent the key entities of the Chinese Wall model:

a) In the Brewer and Nash model, each subject represents a user and any program that might act on the user's behalf. The corresponding NGAC concepts for subject are user and process.
b) NGAC objects are the equivalent of objects in the Chinese Wall model.
c) Instead of object labels, attributes are used to represent conflict-of-interest classes and company datasets, and also the dataset of sanitized objects in a specially designated, conflict-of-interest class.
d) Assignment relations between the aforementioned objects and object attributes capture the relationships in the Chinese Wall model between objects and the datasets in which the objects reside, and between the datasets and conflict-of-interest classes.
e) An association between a user attribute representing all defined users, and an object attribute representing all defined objects and object containers, grants users the initial authorization to read and write any object in any dataset and associated conflict of interest class.

The two principle rules of the Chinese Wall model that also must be addressed by the mapping are the ones for reading and writing objects:

a) Read Rule: a subject s can read object o only if:
   o is in the same dataset as some object previously read by s, or
   o belongs to a conflict-of-interest class for which s has yet to read an object.
b) Write Rule: a subject s can write object o only if:
   s can read o under the read rule, and
   no object can be read, if it is in a dataset different from the one for which write access is requested.

These policy rules, in effect, are separation-of-duty constraints that continually narrow the access rights of a subject as it performs allowed activities. Obligations that effectively capture these rules can be defined. The initial policy configuration would allow a user's process to read and write any object in the data store. As the process accesses objects, obligations are triggered that adjust the policy for both the user and its processes in accordance with the read and write rules. The basic idea is that when a process performs a read access of an object in some conflict-of-interest class, COI_i, its user is denied the ability to read objects in any other dataset in COI_i using different processes. In addition, subsequent read or write attempts of objects in any other dataset in COI_i by this process are denied. However, the user is allowed to employ different processes to read and write to objects in any other dataset in COI_i, in accordance with the read and write rules.

B.2.3 Example mapping

To represent an arbitrary, generic Chinese Wall policy in GOADS, the following key policy elements are needed:

a) a set of object attributes representing each conflict-of-interest class, {COI_1, … , COI_n};
b) a set of object attributes representing each company dataset, {DS_1, … , DS_n}; and
c) a pair of object attributes representing the sanitized dataset, SDS, and the specially designated, conflict-of-interest class, SCOI, which contains it.

As described earlier, assignment relations between the objects and the datasets in which the objects reside, and between the datasets and their respective conflict-of-interest classes depict the principal relationships of the Chinese Wall model.

An example policy configuration depicting two conflict-of-interest classes, each with two company datasets, is illustrated in Figure 2. As mentioned earlier, any users assigned to this policy class are initially authorized full read and write authorization over objects in the data store. In GOADS, users are collectively represented by the user attribute Users, and the data store and its contents are represented similarly by the object attribute Data Store. Users are given authorization to read (r) and write (w) objects in the Data Store through an association, which is depicted as an arc from Users to Data Store, labeled with the associated set of access rights, {r, w}.

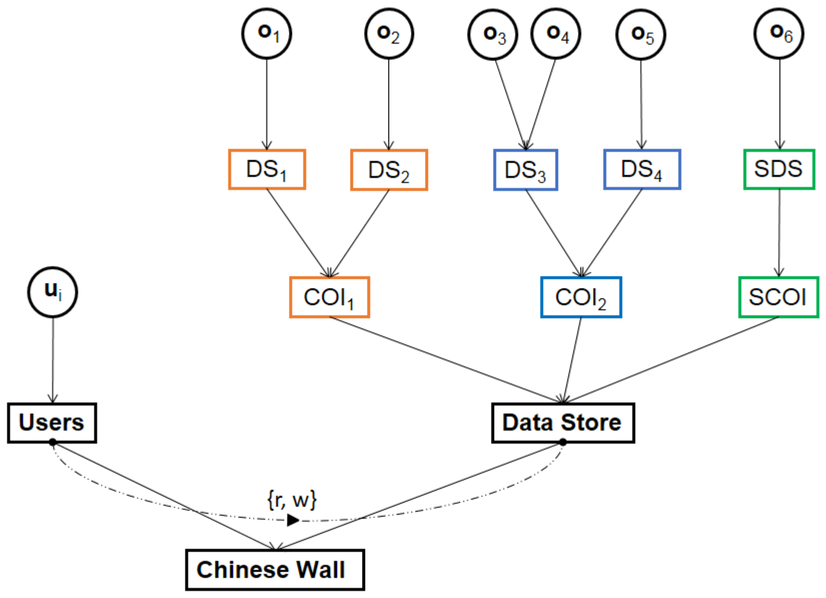

Figure 2: Example Chinese Wall Policy

To capture the Chinese Wall read and write rules, an elaborate obligation must be defined for each COI attribute and any DS attributes contained by the COI attribute. The obligation must also specify the treatment of sanitized objects within the SDS and SCOI. The following template indicates structure of the GOADS' policy expression for the pattern and response components of each obligation, following their respective grammars specified in Annex A:

```
(x, any user of Users performs operation read on any policy element in $COI_i$,
if object getobjectid() in object attribute $DS_j$ then
    deny user process_user(getprocessid()) access right read on elements of intersection of
    policy element $COI_i$, complement of policy element $DS_j$,
    deny process getprocessid() access rights read, write on elements of intersection of
    complement of policy element $DS_j$, complement of policy element SDS,
    deny process getprocessid() access right write on elements of policy element SDS,
if object getobjectid() in object attribute $DS_k$ then
    deny user process_user(getprocessid()) access right read on elements of intersection of
    policy element $COI_i$, complement of policy element $DS_k$,
    deny process getprocessid() access rights read, write on elements of intersection of
    complement of policy element $DS_j$, complement of policy element SDS,
    deny process getprocessid() access right write on elements of policy element SDS,
if object getobjectid() in object attribute $DS_l$ then
    …   )
```

For the example policy configuration illustrated in Figure 2, the following two obligations would be defined as representative of the Chinese Wall read and write rules:

```
(x, any user of Users performs operation read on any policy element in $COI_1$,
if object getobjectid() in object attribute $DS_1$ then
    deny user process_user(getprocessid()) access right read on elements of intersection of
    policy element $COI_1$, complement of policy element $DS_1$,
    deny process getprocessid() access rights read, write on elements of intersection of
    complement of policy element $DS_1$, complement of policy element SDS,
    deny process getprocessid() access right write on elements of policy element SDS,
if object getobjectid() in object attribute $DS_2$ then
    deny user process_user(getprocessid()) access right read on elements of intersection of
    policy element $COI_1$, complement of policy element $DS_2$,
    deny process getprocessid() access rights read, write on elements of intersection of
    complement of policy element $DS_2$, complement of policy element SDS,
    deny process getprocessid() access right write on elements of policy element SDS)

(x, any user of Users performs operation read on any policy element in $COI_2$,
if object getobjectid() in object attribute $DS_3$ then
    deny user process_user(getprocessid()) access right read on elements of intersection of
    policy element $COI_2$, complement of policy element $DS_3$,
    deny process getprocessid() access rights read, write on elements of intersection of
    complement of policy element $DS_3$, complement of policy element SDS,
    deny process getprocessid() access right write on elements of policy element SDS,
if object getobjectid() in object attribute $DS_4$ then
    deny user process_user(getprocessid()) access right read on elements of intersection of
    policy element $COI_2$, complement of policy element $DS_4$,
    deny process getprocessid() access rights read, write on elements of intersection of
    complement of policy element $DS_4$, complement of policy element SDS,
    deny process getprocessid() access right write on elements of policy element SDS)
```

These obligations complete the description of one possible mapping between NGAC and the Chinese Wall policy models.

B.3 Role-based access control

B.3.1 Background

The Role Based Access Control (RBAC) model governs the access of a user to information through roles for which the user is authorized to perform. RBAC is based on several entities: users, roles, permissions, sessions, and objects [RBAC04]. A user represents an individual or an autonomous entity of the system. A role represents a job function or job title that carries with it some connotation of the authority held by a members of the role. Access authorizations on objects are specified for roles, instead of users. A role is fundamentally a collection of permissions to use resources appropriate for carrying out a particular job function, while a permission represents a mode of access to one or more objects that represent the protected resources of a system.

The principle of least privilege requires that a user be given no more privilege than necessary to perform a job. The RBAC model supports this principle through role activation. Users are given authorization to operate in one or more roles, but must utilize a session to gain access to a role. A user may invoke one or more sessions, and each session relates a user to one or more roles. The concept of a session within the RBAC model is equivalent to the more traditional notion of a subject. When a user activates a role to operate within, it acquires the capabilities assigned to the role. Other roles authorized for the user, which have not been activated, remain dormant and the user does not acquire their associated capabilities.

Another important feature RBAC is role hierarchies, whereby one role at a higher level can acquire the capabilities of another role at a lower level, through an explicit inheritance relation (i.e., role x ≥ role y means that role x inherits the permissions of role y). A user assigned to a role within a role hierarchy acquires the capabilities of any roles lower in the hierarchy, as well as those capabilities directly attributed to the role. Standard RBAC also provides features to express policy constraints involving Separation of Duty (SoD) and cardinality. SoD is a security principle used to formulate multi-person control policies in which two or more roles are assigned responsibility for the completion of a sensitive transaction, but a single user is allowed to serve only in some distinct subset of those roles (e.g., not allowed to serve in more than one of two transaction-sensitive roles). Cardinality of a role limits its capacity to a fixed number of users. Cardinality constraints were incorporated into SoD relations in the RBAC standard.

Two types of SoD relations exist: static separation of duty (SSD) and dynamic separation of duty (DSD). SSD relations place constraints on the assignments of users to roles, whereby membership in one role prevents the user from becoming a member of certain other roles, and thereby ensuring the involvement of two or more users in performing a sensitive transaction that requires the capabilities of two or more roles. Dynamic separation of duty relations, like SSD relations, limit the capabilities that are available to a user, while adding operational flexibility, by placing constraints on roles that can be activated within a user's sessions. As such, a user may be a member of two roles in DSD, but unable to execute the capabilities that span both roles within a single session.

B.3.2 Mapping considerations

A formal model exists for RBAC, which can be used to formulate a mapping to its abstractions from those available in the NGAC model. In the RBAC standard, a role essentially represents a collection of users mapped to a collection of permissions [RBAC04]. Each permission in turn represents a collection of object and access right pairs. RBAC user to role mappings can be characterized in GOADS as administrative associations from user attributes uniquely representing each user to user attributes uniquely representing each role. RBAC permission to role mappings can also be characterized in GOADS as associations from the role-representing attributes to the object attributes of each object, through which the appropriate access rights are authorized roles, which are comparable with the RBAC permissions in question. In addition, several other important aspects of policy representation also need to be considered when defining a complete mapping between NGAC and RBAC. They are as follows: compliance with the principle of least privilege, the expression of role hierarchies, and the imposition of Separation of Duty (SoD) constraints.

Role activation plays an essential part in maintaining the principle of least privilege. RBAC allows activation of a subset of the roles assigned to a user via a session, limiting the privileges of the user to those available through its active roles. Least privilege and role activation(\deactivation) concepts can be accommodated in GOADS through the processes acting on behalf of a user and assignments they make(\break) from the user attribute uniquely representing the user to the set of role-representing attributes designated by the administrative associations that characterize permission to role mappings. Obligations are not needed for core RBAC, but for hierarchical RBAC, they can be employed to constrain role activation capabilities conveyed by the administrative associations and enforce the desired behavior.

Assignments between pairs of role-representing attributes in GOADS can be used to represent RBAC role hierarchies in which the attribute at the tail of the assignment inherits the properties of the attribute at the head. Inheritance of properties between role-representing attributes in GOADS are sufficient in and of themselves to capture entirely the property inheritances of the corresponding RBAC roles. One area to be addressed, however, is mitigating the effect of administrative associations that represent user to role mappings, which involve role-representing attributes within a role hierarchy. Since these associations are formed between user attributes and role-representing attributes, hierarchical roles may impart unwanted authorizations over role activation to a user. Prohibitions are used to countermand any unwanted authorizations and render an accurate mapping.

A significant difference exist in the approach used between the NGAC and RBAC models for expressing SoD constraints. In RBAC, both are represented explicitly in the policy configuration via relations, and access enforcement is based on the policy configuration, which remains static. GOADS, however, provides no direct counterpart for representing RBAC SoD constraints explicitly through a static policy configuration. Instead, these properties must be expressed indirectly through the pattern and response components of defined obligations, which dynamically change the policy configuration to enforce these constraints, as both administrative and resource operations are being carried out. That is, SoD constraints are specified indirectly through the pattern and response grammars of obligations defined in the policy configuration, but enforcement of these constraints requires modification of the policy configuration, as their respective obligations are triggered.

To capture a static SoD constraint, an obligation is defined, which is triggered by the creation of an administrative association granting a user authority to create an assignment to a role-representing attribute that is involved in a static SoD role. The obligation’s response checks the conditions of the SoD constraint and when the SoD constraint limit is reached, creates a prohibition that denies the user from involvement in further administrative associations to all other roles involved in the SoD constraint. A complimentary obligation must also be defined, which is triggered whenever an association from a user to a SoD role is removed, and deletes the prohibition originally put in place to block associations to other roles involved in the SoD constraint.

Dynamic SoD constraints are handled similarly to static SoD constraints. An obligation is defined, which is triggered whenever a user activates a dynamic SoD role through administrative privileges assigned to it initially. The obligation’s response checks the conditions of the dynamic SoD constraint and using a prohibition, denies the user from activating other roles involved in this constraint, when the constraint limit is reached. A complimentary obligation must also be defined to undo the above. The obligation is triggered whenever an assignment from a user to a SoD role is removed, and deletes the obligation originally put in place to block activation of other roles involved in the SoD constraint.

B.3.3 Example mapping

B.3.3.1 Constituent analysis

To represent an arbitrary RBAC policy, the following key policy elements and relations in GOADS are needed:

a) a set of user attributes representing each RBAC role, $\{R_1, \dots, R_n\}$, where R_i is the name of the ith role;
b) a set of user attributes representing each RBAC user, $\{U_1, \dots, U_n\}$, where U_i is the name of the ith user;
c) a finite set of object attributes representing each RBAC object, $\{O_1, \dots, O_n\}$, where O_i is the name of the ith object;
d) a set of associations from each RBAC user to the RBAC objects over which it has the authority designated by the access rights, which represents the RBAC privilege assignment relation;
e) a set administrative associations from each RBAC user to the RBAC roles to which it is assigned (granting the administrative access right assign-to), together with a set administrative associations from each RBAC user to itself (granting the administrative access right assign-from), which jointly represent the RBAC user assignment relation;
f) a set of assignments between RBAC roles involved in an RBAC role hierarchy, which represent the inheritance structure of the properties acquired from an inferior role by a superior role; and
g) a set of obligations that constrain the assignment and activation of RBAC roles, such that the allowed behavior of RBAC users complies with RBAC SoD constraints.

An example policy configuration that exhibits the most important features of the RBAC model is illustrated in Figure 3. To keep the example simple only five roles are involved, which could be considered as representative of a larger set of roles for a software development company. Two of the roles, Employee and Contractor, are mutually exclusive of one another; a user can be assigned to either one, but not both. Therefore a static SoD constraint applies to these roles. The remaining roles, Software Engineer, Quality Assurance (QA) Engineer, and Team Member, are involved in a role hierarchy in which the Software and Quality Assurance Engineer roles are each superior to the Team Member role. A dynamic SoD constraint also applies to the two superior roles, preventing a user from being active in both role simultaneously.

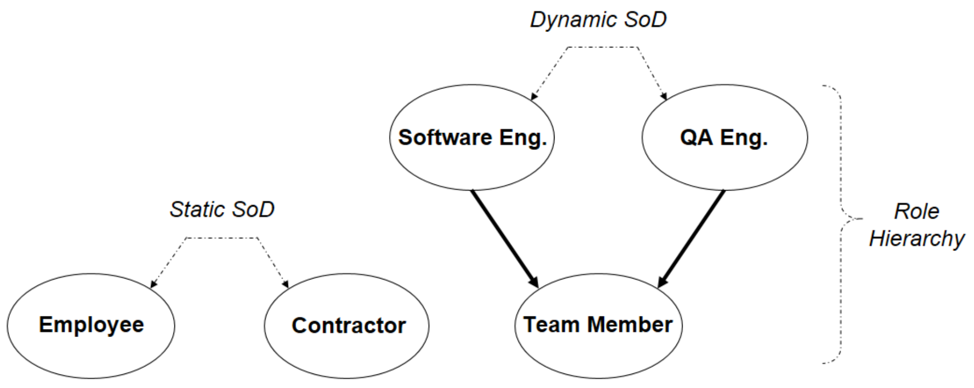

Figure 3: RBAC Policy Configuration

In terms of the permissions allocated to each role, for simplicity, only certain basic objects and operations are included in the example. The Employee role is assigned permissions to access certain company policy documents that are not privy to the Contractor role. A static SoD constraint prevents a user from being assigned to both. New users in either of these roles may initially be assigned to the Team Member role, which allow them to begin reading over requirements, design, and other documentation pertaining to on-going projects. Depending on the skill set, a user is eventually authorized to work in the Software Engineer (SE) role only, in the QA Engineer (QAE) role only, or in both roles non-simultaneously. The SE role allows project code to be produced and modified, while the QAE role allows project code to be written and certified, and test cases to be produced and modified. A static SoD constraint prevents a user from activating both of these roles simultaneously.

For this example, only a single user is discussed. The user in this example is an employee authorized to work as a software engineer on the project team, with no QA engineering responsibilities.

The NGAC equivalent representation of this RBAC policy configuration is complex in comparison, since it contains considerably more detail than that shown in Figure 3. Besides the five RBAC roles and role hierarchy relation, it encompasses the permissions, objects, permission assignment relations, users, and user assignment relations that are often glossed over in an RBAC policy discussion. To better explain how the mapping applies, the details of the GOADS representation of the RBAC policy are decomposed into several segments. The first segment is the GOADS representation of roles and role hierarchies that closely match the RBAC policy depicted in Figure 3 (see B.3.3.2). The second segment is the representation of permissions and the permission assignment relation (see B.3.3.3), followed by the second segment (see B.3.3.4), which is the representation of users and the user assignment relation. The last segment is the representation of RBAC SoD constraints (see B.3.3.5).

B.3.3.2 Roles and role hierarchy

Figure 4 illustrates the first segment of the policy representation in GOADS, which serves as the foundation for the other segments. Three main attribute containers comprise the RBAC policy class: Users, Roles, and Objects. These containers are not essential to the mapping; they are intended as an organizational reference for the reader. Within the Roles container, located at the center of the diagram, are the role-representing attributes for the five RBAC roles. Assignments from the SE and QAE user attributes to that of Team Member represent the equivalent of the role hierarchy relation. The remaining assignments link the Employee, Contractor, and Team Member, user attribute containers to the Role container, and complete this segment.

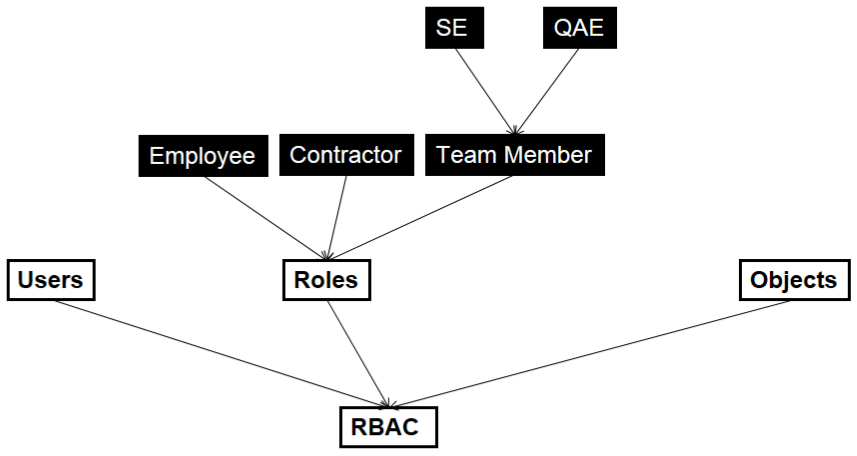

Figure 4: Roles and Role Hierarchy Representation

B.3.3.3 Permission assignment

Figure 5 illustrates the second segment of the policy representation. Several object attributes are specified, which serve as containers for the various categories of information discussed in the RBAC policy description, namely program code (Code), test cases (Tests), code certifications (Certs), documentation (Docs), and company policies (Policies). Associations, represented as arcs from the various role-representing attributes to these containers, assign the authority denoted by the access right set over the containers and their contents. For instance, the association between QAE and Tests grants the QAE role (and any user active within the role) the authority to read (r) and write (w) any objects assigned to Tests. The association also grants the QAE role the same authority over objects assigned to Certs through inheritance. In this fashion, the GOADS association relation is used to realize the RBAC permission assignment relation.

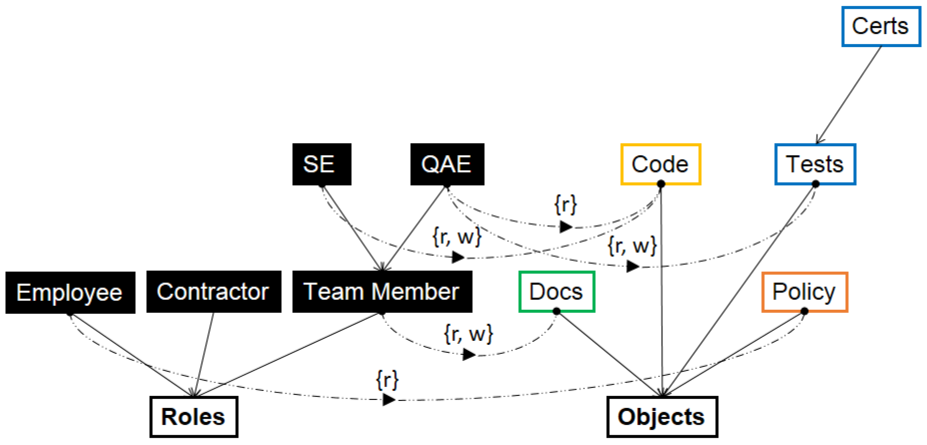

Figure 5: Permission Assignment Representation

B.3.3.4 User assignment

The third segment of the GOADS policy representation is shown in Figure 6. A single user u_i is depicted in the diagram, along with a unique user attribute U_i to which it is assigned. The main purpose of the user attribute is for specifying associations that are applicable to the user. As mentioned earlier, user u_i needs to be able to be able to activate the Employee, Team Member, and SE roles. Therefore, three associations are needed: between U_i and itself, U_i and Employee, and U_i and Team Member. The first association grants authorization for the user to create or delete an assignment from itself to another user attribute over which it holds authorization to complete the assignment operation. The remaining two associations grant authorization for the user to complete the creation or deletion of an assignment respectively to the Employee and Team Member attributes. As before, the associations are represented as downward looping arcs labeled with the appropriate access rights.

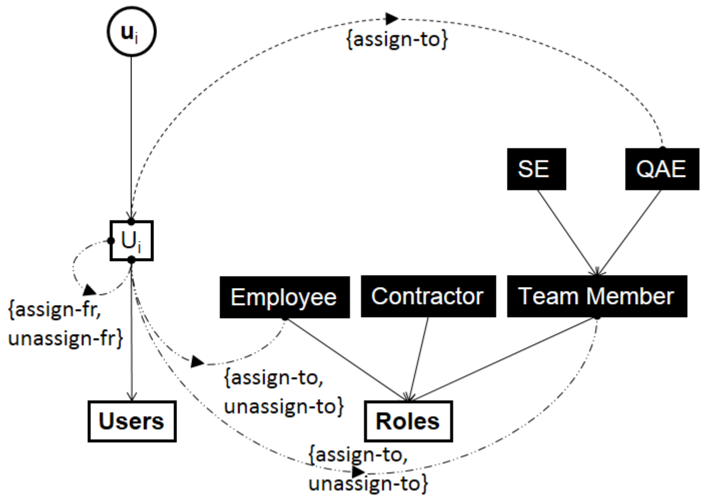

Figure 6: User Assignment Representation

The associations essentially serve as the counterpart to the RBAC user assignment relation. With these associations in place, the user can activate any of the three intended roles (i.e., Employee, Team Member, and SE) as well as the QAE role by creating an assignment from itself to the corresponding role-representing attribute. To prevent the activation of the QAE role by this user, an attribute prohibition is added to the specification, which overrides any attempts to create an assignment between the user u_i and the QAE role. The prohibition is shown as an upward looping arc from the user attribute U_i to the QAE role.

It is through the creation of an assignment to a role-representing attribute that allows the user to activate a role and gain the authority assigned to the role. A process launched by the user to operate on its behalf does so under the user's authority, thus allowing the user to operate within the activated role, as well as any other role previously activated by the user. By deleting an existing assignment between itself and the corresponding role-representing attribute, the user can deactivate a role.

B.3.3.5 SoD constraints

The final area to be addressed in the fourth segment are the two RBAC SoD constraints. As noted earlier, obligations can be created, which serve as counterparts to the SoD constraints. To prevent any user from being assigned (in the RBAC sense of the word) to both the Employee and Contractor roles a pair of obligations are used. One obligation must deny a user from involvement in an association to the Contractor role via a prohibition, whenever an "assign-to" association to the Employee role is created for the user, and the other obligation simply reverses the Contractor and Employee roles in the first obligation. Similarly, if and when the association is deleted from either of these roles (e.g., an employee resigns and is hired by a contractor of the company), the prohibition placed on the other role must be removed through an obligation through a pair of obligations. The following two pairs of obligations captures the static SoD constraint:

(x, **any user of** Users **performs operation** c-assoc **on policy element** Employee,
deny user *process_user*(*getprocessid*()) **access right** assoc-to **on policy element** Contractor)

(x, **any user of** Users **performs operation** c-assoc **on policy element** Contractor,
deny user *process_user*(*getprocessid*()) **access right** assoc-to **on policy element** Employee)

(x, **any user of** Users **performs operation** d-assoc **on policy element** Employee,
delete deny user *process_user*(*getprocessid*()) **access right** assoc-to **on policy element** Contractor)

(x, **any user of** Users **performs operation** d-assoc **on policy element** Contractor,
delete deny user *process_user*(*getprocessid*()) **access right** assoc-to **on policy element** Employee)

The dynamic SoD constraint also requires two pairs of obligations for each role involved in in the constraint. In this case, however, it is the creation of an assignment from any user to the roles in question, SE or QAE, which serves as the trigger for the obligation. As with the static SOD constraint, enforcement is carried out through the creation of a prohibition. The following two pairs of obligations captures the dynamic SoD constraint:

(x, **any user of** Users **performs operation** c-assign **on policy element** SE,
deny user *process_user*(*getprocessid*()) **access right** assign-to **on policy element** QAE)

(x, **any user of** Users **performs operation** c-assign **on policy element** QAE,
deny user *process_user*(*getprocessid*()) **access right** assign-to **on policy element** SE)

(x, **any user of** Users **performs operation** d-assign **on policy element** SE,
delete deny user *process_user*(*getprocessid*()) **access right** assign-to **on policy element** QAE)

(x, **any user of** Users **performs operation** d-assign **on policy element** QAE,
delete deny user *process_user*(*getprocessid*()) **access right** assign-to **on policy element** SE)

These obligations complete the description of one possible mapping between NGAC and the RBAC policy models. The complete policy representation in GOADS for the example RBAC policy is shown in Figure 7. For illustration purposes, the figure also depicts several objects populated within the object containers, which were discussed earlier in the second segment.

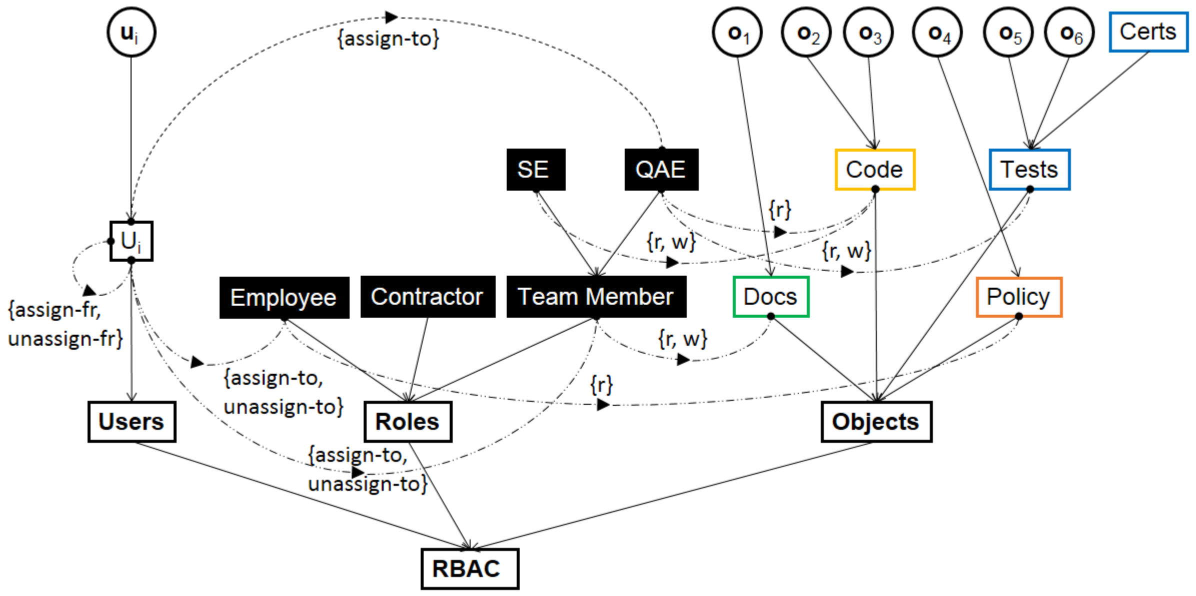

Figure 7: Complete GOADS Policy Representation

Annex C
(Informative)
Bibliography

The following publications are not normative but provide important background for understanding this standard. For information on the current status of the listed document(s), or regarding availability, contact the indicated organization.

BREW89 David Brewer, Michael Nash, The Chinese Wall Security Policy, *IEEE Symposium on Security and Privacy*, May 1989

RBAC04 ANSI INCITS 359-2004, *American National Standard for Information Technology – Role Based Access Control*

SAND92 Ravi S. Sandhu, Lattice-Based Enforcement of Chinese Walls, Computers & Security, Volume 11, Number 8, December 1992, pages 753-763.

Annex D

(Normative)

Summary of Notation

The mathematical notation used in NGAC-GOADS corresponds to a subset of the Z formal specification notation defined in ISO/IEC 13568:2002 (See ZNOT). This annex gives a summary of the notation used and for any differences that exist, their counterpart in the Z notation.

Numbers

$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of natural numbers { 0, 1, 2, ... }
$\mathbb{N}_1$	Set of strictly positive numbers { 1, 2, ... }
$m + n$	Addition
$m - n$	Subtraction
$m * n$	Multiplication
m div n	Division
m mod n	Remainder (modulus)
$m \leq n$	Less than or equal
$m < n$	Less than
$m \geq n$	Greater than or equal
$m > n$	Greater than
$m \ldots n$	Number range

Logic

$\neg p$	Logical negation, not
$p \land q$	Logical conjunction, and
$p \lor q$	Logical disjunction, or
$p \Rightarrow q$	Logical implication
$p \Leftrightarrow q$	Logical equivalence
$\forall x$	Universal quantification
$\exists x$	Existential quantification
$\exists_1 x$	Uniqueness quantification
$\nexists x$	Existential quantification negation (in Z, $\neg\exists$)

Sets and expressions

$x = y$	Equality
$x \neq y$	Inequality
$x \in A$	Set membership
$x \notin A$	Non-membership
$\emptyset$	Empty set
$A \subseteq B$	Set inclusion
$A \subset B$	Strict set inclusion
$\{x_1, \ldots , x_n\}$	Set display
A x B ...	Cartesian product
2^A	Power set (in Z, $\mathbb{P}A$)
$A \cap B$	Set intersection
$A \cup B$	Set union

A \ B	Set difference
first x	First element of an ordered pair
second x	Second element of an ordered pair
#A	Number of elements in a set

Relations

R ⊆ A x B	Binary relation (in Z, R: A ↔ B)
R ⊆ A x B x C	Ternary relation (in Z, R: A ↔ B ↔ C)
dom R	Domain of a relation
ran R	Range of a relation
(a, b)	Ordered pair of a binary relation
(a, b, ... , c)	Ordered tuple of an n-ary relation
a R b	Relation holds between a and b – infix
R(a, b)	Relation holds between a and b – prefix set membership
(a, b) ∈ R	Relation holds between a and b – set membership
R^+	Transitive closure
R*	Reflexive transitive closure

Functions

F(x)	Function application
F ⊆ A x B	Partial functions (in Z, A → B)
F ⊆ A x B	Total functions (in Z, A ⇸ B)

Sequences

⟨⟩	Empty sequence
seq S	Set of finite sequences
seq_1 S	Set of non-empty finite sequences
iseq S	Set of finite injective sequences
$iseq_1$ S	Set of non-empty finite injective sequences (in Z, iseq S \ {⟨⟩})
head S	First element of a sequence
last S	Last element of a sequence
S (i)	i^{th} elment of a sequence
⟨s_1, … , s_n⟩	Sequence display