Open Access
Full Length Article

ORIGINAL ARTICLE

Z Notation Formalization of Blockchain Healthcare Document Sharing Based on CRBAC

Toqeer Ali
toqeer@iu.edu.sa
Islamic University of Madinah, Madinah, Saudi Arabia

ABSTRACT

Healthcare and business solutions are transforming towards decentralized architectures. In this regard, many blockchain-based solutions have been proposed. Some are order-execute architecture while some are based on execute-order architecture of blockchain. Execute-order architectures are famous for general purpose business applications. Hyperledger-composer is a smart-contract framework specifically designed to model business apps comfortably. Hyperledger-composer supports role-based access control, however, it is new to adopt by Healthcare applications. The most important aspect of Healthcare applications is to provide access to sensitive documents. In this paper, we are formally defining constraint role-based access control (CRBAC) in Z formalization language to prove the properties of CRBAC in a Healthcare document sharing for blockchain, specifically for Hyperledger-composer. The paper aims to take Healthcare document's access via constraint-based access control while the interface to the roles and documents are decentralized.

Keyword: Blockchain, Healthcare, Decentralized Healthcare, Access Control, Z formalization

Article Info.
Received: 21 March 2018
Accepted: 23 June 2018
Published: 30 June 2018

INTRODUCTION

Blockchain received great acceptance in last few years. It is a decentralized network of nodes, communicating with each other. All the communication is recorded on a centralized database. The novelty of the database is that, it is hashed chain of transactions that cannot be altered. Moreover, it is also recorded on multiple nodes and record of transactions are stored on the consensus of each nodes. Once a record is saved on blockchain, no one can change or alter it. Any unauthorized change is detectable. This new mechanism brings trust among two untrusted bodies without involving any trusted third party to perform business transaction. The same brings trust

among the patient and Healthcare roles, such as, doctor, technician, laboratories representatives. There are many use-cases where the release of health record needs trust among various stakeholders. Blockchain based solutions help to tackle these problems. One of the IBMs solutions provide Hyperledger composer. Composer is based on role-based access control. For further reading on composer access control, the author refers the reader to the document [9].

In this regard, we are providing a constraint role-based control formalization in Z notation. The formalization is specifically made for Healthcare document sharing on Hyperledger composer among various roles. It will help formally prove all the transactions in a Healthcare record setting for the newly designed Healthcare apps over blockchain. Z formalization is very well defined in [8] for further reading.

Access control is a core functionality required when working with Healthcare. It is required in provision of basic resources needed to certain groups or participants. In [4], the concept of distributed ledger utilized to store transactions. They have ensured that various stakeholders in the e-health have knowledge about rest of parties that are dealing with e-Health resources in order to maintain integrity, scalability, authenticity and audibility. However, the work done in this research deviates from our work in many aspects.

In [7], a cloud based Healthcare system is presented that possess important features necessary for ensuring privacy of Healthcare applications and confidential data. The concept of utilizing clouds for applications has become famous. However, it faces challenges in terms of privacy. In [2] Horizon et. al, employed general data protection regulations (GDPR) to embed multiple security tests and data protection tools in common deployable infrastructure. Study reported in [11] by Zhang et. al, applied various software patterns to address interoperability in Blockchain-based Healthcare Applications. They identified features and implementation challenges in Healthcare interoperability. Given an end-to-end case study of a blockchain-based healthcare application.

Yue et al. [10] developed an application namely Healthcare Data Gateway (HGD) based on blockchain. The solution helps control and share clients data easily without compromising privacy. It provides an excellent way to increase intelligence of Healthcare systems and at the same time keeps patient data private. Their proposed access control model make sure to provide control over Healthcare records in Blockchain network. Dagher et al. [3] presented a framework, a blockchain-based, for providing access to various stackholders in a secure manner. The stackholders includes providers, third parties and patients. The framework is interoperable and work efficiently. The framework preserves the privacy of patients sensitive information. Our framework, named Ancile, utilizes smart contracts in an Ethereum-based blockchain for heightened access control and obfuscation of data, and employs advanced cryptographic techniques for further security.

Contribution: The work done so far provides various aspects of Healthcare records sharing. However, based on our knowledge none of the previous research shown access control mechanisms for Healthcare apps on a private blockchain, such as, Hyperledger Composer. In this paper, we are formally defining in Z Notation all the properties of RBAC and CRBAC of Healthcare document sharing based on SECTET.

Paper Structure: The rest of the paper is organized as follows. Section 2 defines complete RBAC properties in Z formalization. Section 3 shows extension to the NIST formalization of RBAC and that more fits in private blockchains. Section 4 brings constraints to RBAC and explain the document sharing with a role in a Healthcare environment. In section 5 the author concludes the paper.

Specifying RBAC in Z

RBAC is the core access control model of Hyperledger Composer for simplicity of the reader we are initially defining the complete RBAC formalization in Z. Later we will proceed with costraint-based access control for Healthcare document sharing based on SECTET.

2.1 Building Blocks of RBAC

2.1 Building Blocks of RBAC

This is merely an outline of the RBAC model and is described purely in Z notation. For a discussion of the constraints and semantics, see [6].

USERS, ROLES and OPERATIONS are the only basic datatypes in RBAC.

$$[USERS, ROLES, OPERATIONS]$$

We need to define a shorthand symbol ($\curvearrowright$) for *inherits* function since it is used by many other functions and constraints. $r_1 \curvearrowright r_2$ is a relation which returns true if role r_1 inherits from role r_2. We also define the transitive and the transitive, reflexive closure of this relation.

relation $(\_ \curvearrowright \_)$
relation $(\_ \curvearrowright^+ \_)$
relation $(\_ \curvearrowright^* \_)$

$$\_ \curvearrowright \_ , \_ \curvearrowright^+ \_ , \_ \curvearrowright^* \_ : ROLES \times ROLES$$

$$\forall x, y : ROLES \bullet x \curvearrowright y \Leftrightarrow (x, y) \in inherits$$
$$\forall x, y : ROLES \bullet x \curvearrowright^+ y \Leftrightarrow y \in \bigcup \{n : \mathbb{N} \mid n \geq 1 \bullet inherits^n(x)\}$$
$$\forall x, y : ROLES \bullet x \curvearrowright^* y \Leftrightarrow x \curvearrowright^+ y \lor x = y$$

Some helper functions:

$$authorizedRoles : USERS \pfun \mathbb{P}\, ROLES$$

$$\forall u : USERS \bullet authorizedRoles(u) \Leftrightarrow \{r : ROLES \mid \{\exists p : ROLES \mid p \in assignedRoles(u) \land p \curvearrowright^* r\}\}$$

$$authorizedUsers : ROLES \pfun \mathbb{P}\, USRERS$$

$$\forall r : ROLES \bullet authorizedUsers(r) \Leftrightarrow \{u : USERS \mid r \in authorizedRoles(u)\}$$

We define some properties that must hold for a State to be consistent.

$P1 == \forall r : ROLES \bullet \#(authorizedUsers(r)) \leq cardinality(r)$
$P2 == \forall r : ROLES \bullet \neg(r \frown^+ r)$
$P3 == \forall u : USERS;\ r1, r2 : ROLES \bullet r1 \in assignedRoles(u) \land r1 \in assignedRoles(u) \Rightarrow \neg(r1 \frown^+ r2)$
$P4 == \forall u : USERS;\ r1, r2 : ROLES \bullet r1 \in authorizedRoles(u) \land r2 \in authorizedRoles(u) \Rightarrow (r1, r2) \notin ssd$
$P5 == \forall r : ROLES \bullet (r, r) \notin ssd$
$P6 == \forall r1, r2 : ROLES \bullet (r1, r2) \in ssd \Rightarrow (r2, r1) \in ssd$
$P7 == \forall r1, r2 : ROLES \bullet r1 \frown^+ r2 \Rightarrow (r1, r2) \notin ssd$
$P8 == \forall r, r1, r2 : ROLES \bullet r \frown^+ r1 \land r \frown^+ r2 \Rightarrow (r1, r2) \notin ssd$
$P9 == \forall r, r1, r2 : ROLES \bullet r \frown^+ r1 \land (r1, r2) \in ssd \Rightarrow (r, r2) \in ssd$
$P10 == \forall u : USERS \bullet activeRoles(u) \subseteq authorizedRoles(u)$
$P11 == \forall u : USERS;\ r1, r2 : ROLES \bullet r1 \in activeRoles(u) \land r2 \in activeRoles(u) \Rightarrow (r1, r2) \notin dsd$
$P12 == \forall r1, r2 : ROLES \bullet (r1, r2) \in dsd \Rightarrow (r1, r2) \notin ssd$
$P13 == \forall r1 : ROLES \bullet (r, r) \notin dsd$
$P14 == \forall r1, r2 : ROLES \bullet (r1, r2) \in dsd \Rightarrow (r2, r1) \in dsd$
$P15 == \forall r1, r2 : ROLES \bullet r1 \frown^+ r2 \Rightarrow (r1, r2) \notin dsd$
$P16 == \forall r, r1, r2 : ROLES \bullet r \frown^+ r1 \land r \frown^+ r2 \Rightarrow (r1, r2) \notin dsd$
$P17 == \forall r, r2, r2 : ROLES \bullet r1 \frown^+ r1 \land (r1, r2) \in dsd \Rightarrow (r, r2) \in dsd$

2.2 State in RBAC

2.2 State in RBAC

Now, let's define what a State is in RBAC.

State

$users : \mathbb{P}\ USERS$
$roles : \mathbb{P}\ ROLES$
$assignedRoles : USERS \pfun \mathbb{P}\ ROLES$
$activeRoles : USERS \pfun \mathbb{P}\ ROLES$
$inherits : ROLES \leftrightarrow ROLES$
$ssd : ROLES \leftrightarrow ROLES$
$dsd : ROLES \leftrightarrow ROLES$
$cardinality : ROLES \pfun \mathbb{N}$

$users = \mathsf{dom}\ assignedRoles$
$users = \mathsf{dom}\ activeRoles$
$roles = \mathsf{dom}\ cardinality$
$P1 \land P2 \land P3 \land P4 \land P5 \land P6 \land P7 \land P8 \land P9 \land P10 \land P11 \land P12 \land P13 \land P14 \land P15 \land P16 \land P17$

For a complete description of these constraints for consistency of a State, please refer to [6], Section 3.4.

2.3 Operations on State

2.3 Operations on State

addUser

$\Delta State$
$user? : USERS$

$user? \notin users$
$users' = users \cup \{user?\}$
$activeRoles' = activeRoles \cup \{user? \mapsto \varnothing\}$
$assignedRoles' = assignedRoles \cup \{user? \mapsto \varnothing\}$

rmUser

$\Delta State$
$user? : USERS$

$user? \in users$
$users' = users \setminus \{user?\}$
$activeRoles' = \{user?\} \rhd activeRoles$
$assignedRoles' = \{user?\} \rhd assignedRoles$

addRole

$\Delta State$
$role? : ROLES$

$role? \notin roles$
$roles' = roles \cup \{role?\}$
$cardinality' = cardinality \cup \{role? \mapsto \infty\}$

rmRole

$\Delta State$
$role? : ROLES$

$role? \in roles$
$\forall u : USERS \bullet role? \notin assignedRoles(u)$
$\neg(\exists p : ROLES \bullet p \curvearrowright role? \vee role? \curvearrowright p)$
$\neg(\exists p : ROLES \bullet (p, role?) \in ssd)$
$\neg(\exists p : ROLES \bullet (p, role?) \in dsd)$
$roles' = roles \setminus \{role?\}$
$cardinality' = \{role?\} \lhd cardinality$

addAssignment

$\Delta State$
$user? : USERS$
$role? : ROLES$

$user? \in users$
$role? \in roles$
$role? \notin authorizedRoles(user?)$
$\forall p : ROLES \bullet role? \curvearrowright^+ p \Rightarrow p \notin assignedRoles(user?)$
$\forall p : ROLES \bullet p \in assignedRoles(user?) \Rightarrow (p, role?) \notin ssd$
$\forall p : ROLES \bullet role \curvearrowright^* p \Rightarrow \#(authorizedUsers(p)) < cardinality(p)$
$assignedRoles' = (assignedRoles \setminus \{user? \mapsto assignedRoles(user?)\}) \cup \{user? \mapsto (assignedRoles(user?) \cup role?)$

rmAssignment

$\Delta State$
$user? : USERS$
$role? : ROLES$

$user? \in users$
$role? \in roles$
$role? \in assignedRoles(user?)$
$\forall r : ROLES \bullet r \in activeRoles(user?) \wedge role? \curvearrowright^* r \Rightarrow \exists p : ROLES \bullet p \neq role? \wedge p \curvearrowright^+ r \wedge p \in assignedRoles(user?)$
$assignedRoles' = (\{user?\} \lhd assignedRoles) \cup \{user? \mapsto (assignedRoles(user?) \setminus role?)\}$

Before removing a role from assignedRoles, you need to make sure that there is another role which is assigned so that it can take care of the currently activeRoles of the user. Otherwise, the user will be in activeRoles even though he's not authorized by any role.

addInheritance

$\Delta State$
$r1? : ROLES$
$r2? : ROLES$

$r1? \in roles \wedge r2 \in roles$
$r1? \neq r2?$
$\neg(r1? \curvearrowright^+ r2?) \wedge \neg(r2? \curvearrowright^+ r1?)$
$\forall u : USERS;\ r : ROLES \bullet r2? \curvearrowright^+ r \wedge r1? \in authorizedRoles(u) \Rightarrow r \notin assignedRoles(u)$
$\forall r : ROLES \bullet (r, r2?) \in ssd \Rightarrow (r, r1?) \in ssd$
$\forall r : ROLES \bullet (r, r2?) \in dsd \Rightarrow (r, r1?) \in dsd$
$\forall r : ROLES \bullet r2? \curvearrowright^* r \Rightarrow \#(authorizedUsers(r1?) \cup authorizedUsers(r)) \leq cardinality(r)$
$inherits' = inherits \cup (r1?, r2?)$

rmInheritance

$\Delta State$
$r1? : ROLES$
$r2? : ROLES$

$r1? \in roles \land r2 \in roles$
$r1? \curvearrowright r2?$
$\forall u : USERS;\ r : ROLES \bullet$
$\quad u \in authorizedUsers(r1?) \land r \in activeRoles(u)$
$\land\ r2? \curvearrowright^* r \Rightarrow \quad \exists p : ROLES;$
$newinherits : ROLES \leftrightarrow ROLES \bullet$
$\quad p \in assignedRoles(u) \land$
$\quad newinherits = inherits \setminus (r1?, r2?) \land$
$\quad (r \in \bigcup \{n : \mathbb{N} \mid n \geq 1 \bullet newinherits^n(p)\}$
$inherits' = inherits \setminus (r1?, r2?)$

addSsd

$\Delta State$
$r1? : ROLES$
$r2? : ROLES$

$r1? \in roles \land r2? \in roles$
$r1? \neq r2?$
$(r1?, r2?) \notin ssd$
$(r1?, r2?) \notin dsd$
$\forall r : ROLES \bullet r \curvearrowright r1? \Rightarrow (r, r2?) \in ssd$
$\forall r : ROLES \bullet r \curvearrowright r2? \Rightarrow (r, r1?) \in ssd$
$\forall u : USERS \bullet \{r1?, r2?\} \not\subseteq assignedRoles(u)$
$ssd' = ssd \cup \{(r1?, r2?), (r2?, r1?)\}$

rmSsd

$\Delta State$
$r1? : ROLES$
$r2? : ROLES$

$r1? \in roles \land r2? \in roles$
$(r1?, r2?) \in ssd$
$\forall r : ROLES \bullet r1? \curvearrowright r \Rightarrow (r, r2?) \notin ssd$
$\forall r : ROLES \bullet r2? \curvearrowright r \Rightarrow (r, r2?) \notin ssd$
$ssd' = ssd \setminus \{(r1?, r2?), (r2?, r1?)\}$

addDsd

$\Delta State$
$r1? : ROLES$
$r2? : ROLES$

$r1? \in roles \land r2? \in roles$
$r1? \neq r2?$
$(r1?, r2?) \notin ssd$
$(r1?, r2?) \notin dsd$
$\forall r : ROLES \bullet r \curvearrowright r1? \Rightarrow (r, r2?) \in dsd$
$\forall r : ROLES \bullet r \curvearrowright r2? \Rightarrow (r, r1?) \in dsd$
$\forall u : USERS \bullet \{r1?, r2?\} \not\subseteq activeRoles(u)$
$dsd' = dsd \cup \{(r1?, r2?), (r2?, r1?)\}$

rmDsd

$\Delta State$
$r1? : ROLES$
$r2? : ROLES$

$r1? \in roles \land r2? \in roles$
$(r1?, r2?) \in dsd$
$\forall r : ROLES \bullet r1? \curvearrowright r \Rightarrow (r, r2?) \notin dsd$
$\forall r : ROLES \bullet r2? \curvearrowright r \Rightarrow (r, r2?) \notin dsd$
$dsd' = dsd \setminus \{(r1?, r2?), (r2?, r1?)\}$

setCardinality

$\Delta State$
$role? : ROLES$
$c? : \mathbb{N}$

$role? \in roles$
$\#(authorizedUsers(role?)) \leq c$
$cardinality' = (\{role?\} \lhd cardinality) \cup \{role? \mapsto c\}$

addActiveRoles

$\Delta State$
$user? : USERS$
$roles? : \mathbb{P}\, ROLES$

$user? \in users$
$roles? \in \mathbb{P}\, roles$
$roles? \subseteq authorizedRoles(user?)$
$\forall r1, r2 : ROLES \bullet \{r1, r2\} \subseteq (activeRoles(user?)$
$\cup roles?) \Rightarrow (r1, r2) \notin dsd$
$activeRoles' = (user? \lhd activeRoles) \cup$
$\{user? \mapsto (activeRoles(user?) \cup roles?)\}$

rmActiveRoles

$\Delta State$
$user? : USERS$
$roles? : \mathbb{P}\, ROLES$

$user? \in users$
$roles? \subseteq activeRoles(user?)$
$activeRoles' = (user? \lhd activeRoles) \cup \{user?$
$\mapsto (activeRoles(user?) \setminus roles?)\}$

MODIFICATIONS TO NIST

The basic NIST RBAC paper [6] does not define what ROLES are. It treats them as basic type and leaves it at that. For our purposes, they are used to define what operations a role is allowed to perform. This feature is added to *State* by means of the partial function *allowedOps*. We call the modified state, *MState*.

MState

$State$
$operations : \mathbb{P}\, OPERATIONS$
$allowedOps : ROLES \nrightarrow \mathbb{P}\, OPERATIONS$

$roles = \mathsf{dom}\ allowedOps$

Now we define how we can add and remove operations and allowed operations to and from an *MState*.

addOp

$\Delta MState$
$op? : OPERATIONS$

$op? \notin operations$
$operations' = operations \cup \{op?\}$

When removing an operation, we have to make sure that all the roles that are allowed to perform that operation are updated.

rmOp

$\Delta MState$
$op? : OPERATIONS$

$op? \in operations$
$operations' = operations \setminus \{op?\}$
$allowedOps' = \{r : ROLES \bullet$
$\{r \mapsto (allowedOps(r?) \setminus \{op?\})\}\}$

addAllowedOps

$\Delta MState$
$ops? : \mathbb{P}\, OPERATIONS$
$role? : ROLES$

$role? \in roles$
$ops? \nsubseteq allowedOps(role?)$
$allowedOps' = (role? \lhd allowedOps) \cup$
$\{role? \mapsto (allowedOps(role?) \cup ops?)\}$

rmAllowedOps

$\Delta MState$
$ops? : \mathbb{P}\, OPERATIONS$
$role? : ROLES$

$role? \in roles$
$ops? \subseteq allowedOps(role?)$
$allowedOps' = (role? \lhd allowedOps) \cup \{role?$
$\mapsto (allowedOps(role?) \setminus ops?)\}$

3.1 Deciding Validity

3.1 Deciding Validity

A user is allowed to perform an operation if there is a role which is allowed to perform that operation and the user belongs to active users of that role. We define a syntax for using this function. $o \models u$ (read as *o is valid for u*) iff user u is allowed to perform the operaion o.

relation $\_ \models \_$

$$\_ \models \_ : OPERATIONS \times USERS$$
$$\forall o : OPERATIONS;\ u : USERS \bullet o \models u \Leftrightarrow \{\exists r : ROLES \mid r \in roles \bullet o \in allowedOps(r) \land r \in activeRoles(u)\}$$

The initialization of *State* is given by the axiomatic definition. We only need to define *assignedRoles*, *users* and *roles* as an empty set. All the other sets and relations depend on the existence of one of these due to properties $P1 \ldots P17$ and *State* constraints. They are, therefore, automatically initialized to empty sets.

$$InitState == \{assignedRoles = \varnothing, users = \varnothing, roles = \varnothing\}$$

In the next section, we add the definitions which enable us to use CRBAC in addition to RBAC.

ADDING CONSTRAINTS

RBAC, in its basic form, does not cater for the principle of least previledge. Constrained based RBAC (CRBAC) is one approach developed by Alam et al. [1] to add partial inheritance of permissions and for using RBAC in an SOA architecture. (Not a very good description.)

Since the architecture of CRBAC depends on the Document Model of SECTET for the definition of constraints, we need to define the document model before we can define CRBAC.

5.1 Document Model

The SECTET architecure is based on UML. The Document Model describes different entities and how they are associated with each other. It includes different constructs such as association and generalization. Please refer to Figure 4. in [**?**] for an example of a Document Model.

This specification of the Document Model is based on the formalization of UML by Evans [5]. First, we need to define a class in terms of its attributes, and methods. Values will be useful when we define instantiation of classes to objects. Names are used to define roles of classes in associations and the names of associations themselves.

$$[ATTRIBUTES, NAME, ARGS]$$

$$BASICTYPE ::= nil \mid DATE \mid TIME \mid true \mid false \mid \mathbb{N} \mid \mathbb{Z}$$
$$METHODS == (NAME \times \mathbb{P}\, ARGS) \pfun$$
$$BASICTYPEARGS == \Theta$$

CLASS

$$attribs : \mathbb{P}\, ATTRIBUTES$$
$$methods : \mathbb{P}\, METHODS$$

Association between classes is defined using *associations* and *association ends*. An association end describes the role of the class in the association and an association is composed of a name and two association ends.

ASSOCIATONEND

$rolename : NAME$
$class : CLASS$
$mul : \mathbb{P}\,\mathbb{N}$

ASSOCIATION

$name : NAME$
$e_1, e_2 : ASSOCIATIONEND$

$e_1.rolename \neq e_2.rolename$

Now, we can define the document model precisely.

relation $\_ \sqsubset \_$
relation $\_ \sqsubset^+ \_$
relation $\_ \sqsubset^* \_$

DOCUMENTMODEL

$classes : \mathbb{P}\ CLASS$
$assocs : \mathbb{P}\ ASSOCIATION$
$superclass : CLASS \nrightarrow CLASS$

$\forall\, c_1, c_2 : CLASS \bullet superclass(c_2) = c_1 \Rightarrow c_1 \sqsubset c_2$
$\forall\, c : CLASS \bullet \neg(c \sqsubset^+ c)$

We remove the possibility of circular inheritance using the second constraint.

Class inheritance is defined using an axiom. This states that all sub classes inherit all attributes and functions of the super class. This is all that is needed to define inheritance. $c_1 \sqsubset c_2$ means that c_1 is a super class of c_2. Of course, as usual, we'll need to define the reflexive, transitive closure along with the definition.

$\_ \sqsubset \_, \_ \sqsubset^+ \_, \_ \sqsubset^* \_ : CLASS \times CLASS$

$\forall\, c_1, c_2 : CLASS \bullet c_1 \sqsubset c_2 \Rightarrow$
$\quad \forall\, a : ATTRIBUTES \bullet a \in c_1.attribs$
$\quad\quad \Rightarrow a \in c_2.attribs$
$\quad \land\, \forall\, m : METHODS \bullet m \in c_1.methods$
$\quad \Rightarrow m \in c_2.methods$
$\forall\, c_1, c_2 : CLASS \bullet c_1 \sqsubset^+ c_2 \Rightarrow$
$\quad c_1 \in \bigcup \{n : \mathbb{N} \mid n \geq 1 \bullet superclass^n(c_2)\}$
$\forall\, c_1, c_2 : CLASS \bullet c_1 \sqsubset^* c_2 \Rightarrow$
$\quad c_1 \sqsubset^+ c_2 \lor c_1 = c_2$

This takes care of the instances of the classes as well as instances just assign values to attributes.

To define objects, we need to define two constructs. One is a function which returns the objectID – a unique identifier which identifies an object of a class – and the other is links which gives links between instances of class.

$[ObjID]$
$OBJECTS == CLASS \nrightarrow \mathbb{P}\ ObjID$

Instantiation

$obj : OBJECTS$
$links : ASSOCIATION \nrightarrow (ObjID \times ObjID)$
$values : (ObjID \times ATTRIBUTES) \nrightarrow BASICTYPE$
$objClass : ObjID \rightarrow CLASS$

$\forall\, o : ObjID;\ c : CLASS \bullet o \in obj(c) \Rightarrow$
$\quad objClass(o) = c$
$\forall\, as : ASSOCIATION;\ o_1 : ObjID, o_2 : ObjID$
$\quad \bullet (o_1, o_2) \in$
$links(as) \Rightarrow$
$\quad as.e1.class = objClass(o_1) \land as.e2.class = objClass(o_2)$

The first constraint is self-explanatory and defines what objClass is. The second one means that a link can only be created between two objects if there is an association between the classes of the objects.

5.1.1 Minor Variables and Functions

Before using CRBAC state, we need to define a global variable which can return the value of the current user logged making the request. We call this the *requestingUser*. This value will be initialized by the system when a request is made. We also define a method for returning the *objID* of the object of a class created for the user making the request. *Both of these operations are implementation dependant and cannot be described here.*

$$
\begin{array}{|l}
requestingUser : USERS \\
userObj : (USERS \times CLASS) \pfun ObjID
\end{array}
$$

The function *pickOne*, when given a set of *BASICTYPE*, returns a single element belonging to the set. If the set contains more than one elements, the function returns *false*. (It is not however called for sets which have more than one elements.)

$$
\begin{array}{|l}
pickOne\ \_ : \mathbb{P}\, BASICTYPE \fun BASICTYPE \\
\hline
\forall x : \mathbb{P}\, BASICTYPE;\ y : BASICTYPE \mid \#x = 1 \\
\land\ y \in x \bullet pickOne(x) = y \\
\forall x : \mathbb{P}\, BASICTYPE;\ y : BASICTYPE \mid \#x \neq \\
1 \bullet pickOne(x) = false
\end{array}
$$

5.1.2 Redefining Operations

In SECTET , constraints are dependant on the values passed to operations. So, we cannot leave operations as basic types as in RBAC. We redefine operations as such:

$$
\begin{array}{|l}
\hline
OPERATION \\
\hline
name : NAME \\
params : \mathbb{P}\, NAME \\
\hline
\end{array}
$$

An operation has a name and a list of parameters. An operation call assigns all values of the arguments to some basic type. These are the arguments passed to the operation during the call.

$$
\begin{array}{|l}
\hline
OPCALL \\
\hline
op : OPERATION \\
vals : NAME \fun BASICTYPE \\
\hline
op.params = \mathsf{dom}\ vals \\
\hline
\end{array}
$$

5.2 Defining CRBAC expressions

CRBAC adds *access constraints* to the basic RBAC model.

CRBAC describes expressions using a predicate language called SECTET-PL. A SECTET-PL expression (Ψ) is of two types: *navigation expression* (Θ) and *predicate expression*(Φ).

$$
\begin{array}{|l}
\Psi ::= \Theta \mid \Phi \\
\Theta ::= BASICTYPE \mid ObjID \mid val\langle\!\langle ObjID \times \\
ATTRIBUTES \rangle\!\rangle \mid opval\langle\!\langle NAME \rangle\!\rangle \\
\quad \mid selectOne\langle\!\langle ObjID \times NAME \times \Phi \rangle\!\rangle \\
\quad \mid select\langle\!\langle ObjID \times NAME \times \Phi \rangle\!\rangle \mid \\
\qquad funCall\langle\!\langle ObjID \times NAME \times \mathbb{P}\, ARGS \rangle\!\rangle \\
\quad \mid subjectMap\langle\!\langle CLASS \rangle\!\rangle \\
\Phi ::= true \mid false \mid or\langle\!\langle \Phi \times \Phi \rangle\!\rangle \mid and\langle\!\langle \Phi \times \Phi \rangle\!\rangle \mid \\
\qquad neg\langle\!\langle \Phi \rangle\!\rangle \mid eq\langle\!\langle \Psi \times \Psi \rangle\!\rangle \mid ls\langle\!\langle \Theta \times \Theta \rangle\!\rangle \mid \\
\qquad gr\langle\!\langle \Theta \times \Theta \rangle\!\rangle \mid leq\langle\!\langle \Theta \times \Theta \rangle\!\rangle \mid geq\langle\!\langle \Theta \times \Theta \rangle\!\rangle \\
\qquad \mid let\langle\!\langle VAR \times \Theta \times \Phi \rangle\!\rangle
\end{array}
$$

The *NAME* in both *select* and *selectOne* refers to the name of the association end that the selection is referring to. *NAME* in funcall refers to the name of the function being called.

Where $\lor, \land, \neg, <, >, \leq, \geq$ have the same meanings as their numerical and logical counterparts.

5.2.1 Evaluating SECTET Expressions

For the purpose of assigning values to a SECTET expression, we define an operator called called *eval*. For evaluating permission predicates, the relation defined in Section 5.5.2 is used. The evaluation of navigation expressions is done directly by the *eval* function.

relation *eval* _

$eval\ \_ : (\Psi \times CRBACState \times OPCALL) \rightarrow \mathbb{P}\ BASICTYPE$

$\forall\ e : \Phi;\ s : CRBACState;\ c : OPCALL \bullet$
$eval(e, s, c) \Leftrightarrow \{(c, s) \vDash_s e\}$
$\forall\ e : BASICTYPE;\ s : CRBACState;\ c : OPCALL$
$\bullet\ eval(e, s, c) = \{e\}$
$\forall\ e : ObjID;\ s : CRBACState;\ c : OPCALL$
$\bullet\ eval(e, s, c) = \{e\}$
$\forall\ o : ObjID;\ a : ATTRIBUTES;\ s : CRBACState$
$: c : OPCALL \bullet$
$eval(val(o, a), s, c) = \{s.values(o, a)\}$
$\forall\ c : OPCALL;\ n : NAME;\ s : CRBACState \bullet$
$eval(opval(n), s, c) = \{c.vals(n)\}$
$\forall\ o : ObjID;\ n : NAME;\ p : \Phi;\ s : CRBACState$
$: c : OPCALL \bullet \quad eval(select(o, n, p), s, c) =$
$\{b : ObjID \mid$
$b \in s.obj$
$\wedge\ (\exists\ as : ASSOCIATION \bullet$
$((o, b) \in s.links(as) \wedge as.e2.rolename$
$= n) \vee$
$((b, o) \in s.links(as) \wedge as.e1.rolename$
$= n)$
$)$
$\wedge\ (c, s) \vDash_s p$
$\}$
$\forall\ o : ObjID;\ a : NAME;\ p : \Phi;\ s : CRBACState$
$: c : OPCALL \bullet$
$eval(selectOne(o, a, p), s, c) =$
$\{pickOne(eval(select(o, a, p), s, c))\}$
$\forall\ o : ObjId;\ n : NAME;\ as : \mathbb{P}\ ARGS;\ s :$
$CRBACState;\ c : OPCALL \bullet$
$eval(funcall(o, n, as), s, c) =$
$\{s.objClass(o).methods(n, \bigcup\{a :$
$\Psi \mid a \in as \bullet eval(a, s, c)\})\}$
$\forall\ e : CLASS;\ s : CRBACState;\ c : OPCALL$
$\bullet\ eval(subjectMap(e), s, c) =$
$\{userObj(requestingUser, e)\}$

The evaluations are self explanatory but require some clarifications at a few points.

$eval(val(o, a), s, c)$ returns the value assigned to attribute a of object o in CRBACState s using the function *values* in s.

$eval(opval(n), s, c)$ returns the value passed by the user during the opcall c for the variable n. This is given by the *vals* function of opcall c.

In *select*, the predicate specifies that the evaluation of *select* navigation expression yeilds all objects which are linked to the object passed with an instantiation of the association end name passed. Specifically, all objects such that there exists an association which links them with the original object with the association end name passed to *select*. It also ensures that the predicate expression yields true.

$eval(funcall(o, n, as), s, c)$ returns the value computed by calling method n with arguments *as* of object o in CRBACState s. It first evaluates all navigation expressions passed as arguments and combines the results in a set and passes it to the method n. The function *methods* in every class maps a method name and arugments to it's value. This function is used here for computation of final result of the function call.

$eval(subjectMap(e), s, c)$ returns the object of a class corresponding to the calling user. This is implemented through the minor functions described in Section 5.1.1

5.3 Defining PACs

A PAC is defined as a collection of operation, role and a predicate expression.

PAC
$op : OPERATION$
$role : ROLE$
$pred : \Phi$

The three types of PAC are defined using this same definition. Their semantics are made clear from their use.

5.4 CRBAC State

Now, the *MState* can be refined using DOCUMENTMODEL, Instantiation and PACs for CRBAC. We call this new state, the *CRBACState*.

CRBACState

$MState$
$DOCUMENTMODEL$
$Instantiation$
$gPACs : \mathbb{P}\, PAC$
$ddPACs : \mathbb{P}\, PAC$
$nPACs : \mathbb{P}\, PAC$

$\forall p : PAC \bullet p \notin ((ddPACs \cap nPACs) \cup (ddPACs \cap gPACs) \cup (gPACs \cap nPACs))$

The only constraint for PACs is that a single PAC cannot be of any *two* types.

We define some operations for addition and removal of PACs from a state.

addGPAC

$\Delta CRBACState$
$p? : PAC$

$p? \notin gPACs$
$gPACs' = gPACs \cup \{p?\}$

We do not need to check if $p?$ is already contained in another PAC collection because we already have this constraint in the basic *CRBACState* definition.

rmGPAC

$\Delta CRBACState$
$p? : PAC$

$p? \in gPACs$
$gPACs' = gPACs \setminus \{p?\}$

Similar operations are defined for *ddPAC* and *nPAC*:

addDDPAC

$\Delta CRBACState$
$p? : PAC$

$p? \notin ddPACs$
$ddPACs' = ddPACs \cup \{p?\}$

rmDDPAC

$\Delta CRBACState$
$p? : PAC$

$p? \in ddPACs$
$ddPACs' = ddPACs \setminus \{p?\}$

addNPAC

$\Delta CRBACState$
$p? : PAC$

$p? \notin nPACs$
$nPACs' = nPACs \cup \{p?\}$

rmNPAC

$\Delta CRBACState$
$p? : PAC$

$p? \in nPACs$
$nPACs' = nPACs \setminus \{p?\}$

5.5 Defining Satisfiability and Validity

In order to determine which operations are allowed to which users, we need to define two operators.

5.5.1 Constrained Validity

We defined the basic validity operator $\models$ in Section 3.1 . Here we define *constrained validity* using the operator $\models_c$. We say that *an operation o is valid with constrains in state s for a user u* iff

1. the user belongs to a role which is in activeRoles

of that operation,

2. there is at least one general or data dependant constraint allowing that operation to that role and

3. there is no negative constraint prohibiting that role from performing that operation

This is written as $(o, s) \models_c u$. Specifically,

relation $\_ \models_c \_$

$$
\begin{array}{|l}
\_ \models_c \_ : (OPCALL \times CRBACState) \times USERS \\
\hline
\forall\, o : OPCALL, s : CRBACState;\ u : USERS \bullet \\
\quad (o, s) \models_c u \Leftrightarrow \\
\quad (\exists\, r : ROLES \mid r \in s.roles \bullet \\
\qquad o.op \in allowedOps(r) \land r \in \\
\qquad\quad activeRoles(u) \\
\qquad \land\, (\exists\, p : PAC \mid p.op = o.op \land p.role = r \\
\qquad \bullet\, (p \in gPACs \lor p \in ddPACs) \land (o, s) \\
\qquad \models_s p.pred) \\
\qquad \land\, (\neg \exists\, p : PAC \mid p.op = o \land p.role = \\
\quad r \bullet p \in nPACs \land (o, s) \models_s p.pred) \\
\quad)
\end{array}
$$

5.5.2 Satisfiability

A CRBACState s is said to satisfy a predicate p iff the predicate returns the value true when applied to that state. For this purpose, we define the satisfiability operator $\models_s$. If $s \models_s p$, we say that state s satisfies the predicate expression p.

relation $\_ \models_s \_$
relation $\_ \not\models_s \_$

$$
\begin{array}{|l}
\_ \models_s \_ : (OPCALL \times CRBACState) \times \Phi \\
\_ \not\models_s \_ : (OPCALL \times CRBACState) \times \Phi \\
\hline
\forall\, c : OPCALL;\ s : CRBACState;\ p : \Phi \bullet (s, c) \\
\qquad \models_s p \Leftrightarrow p = true \\
\quad \lor \exists\, p_1, p_2 : \Phi \bullet (p = or(p_1, p_2)) \land ((s, c) \models_s p_1 \\
\qquad \lor (s, c) \models_s p_2) \\
\quad \lor \exists\, p_1, p_2 : \Phi \bullet (p = and(p_1, p_2)) \land ((s, c) \models_s p_1 \\
\qquad \land (s, c) \models_s p_2) \\
\quad \lor \exists\, p_0 : \Phi \bullet (p = neg(p_0)) \land \neg((s, c) \models_s p_0) \\
\quad \lor \exists\, p_1, p_2 : \Psi \bullet (p = eq(p_1, p_2)) \land \\
\qquad (pickOne(eval(p_1, s, c)) = \\
\qquad pickOne(eval(p_2, s, c))) \\
\quad \lor \exists\, p_1, p_2 : \Theta \bullet (p = ls(p_1, p_2)) \land \\
\qquad (pickOne(eval(p_1, s, c)) < \\
\qquad pickOne(eval(p_2, s, c))) \\
\quad \lor \exists\, p_1, p_2 : \Theta \bullet (p = gr(p_1, p_2)) \land \\
\qquad (pickOne(eval(p_1, s, c)) > \\
\qquad pickOne(eval(p_2, s, c))) \\
\quad \lor \exists\, p_1, p_2 : \Theta \bullet (p = leq(p_1, p_2)) \land \\
\qquad (pickOne(eval(p_1, s, c)) \leq \\
\qquad pickOne(eval(p_2, s, c))) \\
\quad \lor \exists\, p_1, p_2 : \Theta \bullet (p = geq(p_1, p_2)) \land \\
\qquad (pickOne(eval(p_1, s, c) \\
\qquad) \geq pickOne(eval(p_2, s, c))) \\
\quad \lor \exists\, z_0 : VAR;\ e_0 : \Theta;\ p_0 : \Phi \bullet \\
\qquad (p = let(z_0, e_0, p_0)) \land \\
\qquad ((c, s) \models_s p_0[eval(e_0, s, c)/z_0]) \\
\forall\, s : CRBACState;\ p : \Phi \bullet s \not\models_s p \Leftrightarrow \neg(s \models_s p)
\end{array}
$$

The semantics of these definitions are all evident except for *let*. A *let* statement creates a lexical scope for the variable in a permission predicate expression. We use the default *substitution* syntax of Z to accommodate this. Any appearance of the variable z_0 in permission predicate p_0 is replaced with the value given by evaluation of the navigation expression e_0 passed to the *let* expression. The resulting permission predicate is evaluated as any normal permission predicate.

The use of *pickOne* is necessary in places where the comparison is being made because *eval* returns a set of *BASICTYPE*s. They would contain only a single element which is returned by *pickOne*. Only in *let* is there a possibility (and indeed use) of multiple return values (due to *selet* type navigation expression). This

usage in *let* and *select* is the reason *eval* was defined to return a set in the first place.

CONCLUSION

The paper formally proves the properties of RBAC and CRBAC in Z Notation and further shows how to share Healthcare documents using SECTET. The reason to show this formalization is due to the new era of decentralized network apps that brought a radical change in future Healthcare and Business apps. RBAC and CRBAC are the very important components of existing blockchain solutions. However, various architectures are available for blockchain, such as, public and private blockchain. CRBAC is required in private blockchain because a user in private blockchain shows his/her identity before getting eligibility to become part of the network.

[1] M. Alam, R. Breu, and M. Hafner. Model-Driven Security Engineering for Trust Management in SECTET. *JOURNAL OF SOFTWARE*, 2(1):47, 2007.

[2] Ed Conley and Matthias Pocs. Gdpr compliance challenges for interoperable health informaon exchanges (hies) and trustworthy research environments (tres). *European Journal for Biomedical Informatics*, 14(3):4861, 2018.

[3] Gaby G Dagher, Jordan Mohler, Matea Milojkovic, and Praneeth Babu Marella. Ancile: Privacy-preserving framework for access control and interoperability of electronic health records using blockchain technology. *Sustainable Cities and Society*, 39:283–297, 2018.

[4] João Pedro Dias, Luís Reis, Hugo Sereno Ferreira, and Ângelo Martins. Blockchain for access control in e-health scenarios. *arXiv preprint arXiv:1805.12267*, 2018.

[5] A.S. Evans. Reasoning with UML class diagrams. *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, pages 102–113, 1998.

[6] S.I. Gavrila and J.F. Barkley. Formal specification for role based access control user/role and role/role relationship management. *Proceedings of the third ACM workshop on Role-based access control*, pages 81–90, 1998.

[7] A. Iyengar, A. Kundu, U. Sharma, and P. Zhang. A trusted healthcare data analytics cloud platform. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1238–1249, July 2018.

[8] Ben Potter, David Till, and Jane Sinclair. *An introduction to formal specification and Z*. Prentice Hall PTR, 1996.

[9] IBM Hyperledger Project. Hyperledger Composer Access Control Language, 2018. [Online; accessed 1st-May-2018].

[10] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. *Journal of medical systems*, 40(10):218, 2016.

[11] Peng Zhang, Jules White, Douglas C Schmidt, and Gunther Lenz. Applying software patterns to address interoperability in blockchain-based healthcare apps. *arXiv preprint arXiv:1706.03700*, 2017.