An Expressive, Flexible and Uniform Logical Formalism for Attribute-based Access Control

Jiaming Jiang
Computer Science Department,
North Carolina State University
Raleigh, North Carolina
Email: jjiang13@ncsu.edu

Rada Chirkova
Computer Science Department,
North Carolina State University
Raleigh, North Carolina
Email: rychirko@ncsu.edu

Jon Doyle
Computer Science Department,
North Carolina State University
Raleigh, North Carolina
Email: Jon_Doyle@ncsu.edu

Arnon Rosenthal
The MITRE Corporation
Bedford, Massachusetts
Email: arnie@mitre.org

***Abstract*—Attribute-based access control (ABAC) is a general access control (AC) model that subsumes numerous earlier AC models. Its increasing popularity stems from the intuitive generic structure of granting permissions based on domain-dependent attributes of users, subjects, objects, and other entities in the system. Multiple formal and informal languages have been developed to express policies in terms of such attributes.**

The utility of ABAC policy languages is potentially undermined without a properly formalized underlying model. The high-level structure in a majority of ABAC models consists of sets and sets of sets, expressions that demand that the reader unpack multiple levels of sets and tokens to determine what things mean. The resulting reduced readability potentially endangers correct expression and reduces maintainability and validation. These problems could be multiplied with models that employ nonuniform representations of actions and their governing policies.

In this paper, we address these problems by recasting the high-level structure of ABAC models in a logical formalism that treats all types of actions uniformly. Our formalism uses a simple variant of description logics to model the high-level structure, and function-free first-order logic with equality to represent and reason about the policies. Use of description logics for model formalizations, including hierarchies of types of entities and attributes, is a promise of improved *usability*, compared with existing ABAC models, in specifying the relationships between and requirements on domain-dependent attributes. Our formal model provides improved *flexibility* in supporting a variety of different requirements depending on the domain. Specifically, we will discuss how to modify the model if time plays a role in authorizing a requested action, if different policies would potentially arrive at conflicting decisions, and if default and exception rules are in application.

***Keywords*—attribute-based access control, description logics, first-order logic, logical formalism, policy language**

I. Introduction

Access control (AC) concerns the problems of designing, expressing, and mechanizing policies that decide whether a party, such as a human *user* or one of a user's *subject* computational processes, has been granted *permissions* sufficient to perform some *action*, such as read or write, on a specific *object*, such as a file in a computer system. As summarized in [1] and surveyed in [2], recent years have seen significant development of *attribute-based access control* (ABAC) models, which provide generality and subsume many AC models by allowing definitions of domain-dependent properties called *attributes*. Translations have shown how one can simulate in ABAC [3], [4] the discretionary access control (DAC), mandatory access control (MAC), and role-based access control (RBAC) models.

Research on formalizing ABAC currently takes two major directions. One direction develops policy languages [5], [6], [7], [8] for use with ABAC models. The other develops means for representing the high-level structure and components of ABAC models and for determining how different components relate to each other [3], [4], [9], [10], [11], [12].

The formal policy language specifications [5], [6], [7], [8], [13] often have a well-defined semantics and a uniform syntax of actions and policies. The underlying model representations of the language specifications, however, do not provide a clear picture of how the main components interact with each other, leading to less readable and more error-prone ABAC modeling, especially during system design and maintenance. For example, hierarchies of users or subjects and of attributes cannot be conveniently and efficiently represented in an authorization policy, and if an attribute is only defined on some subset of the users, administrators have the burden of making sure the other users do not have a value for that attribute.

On the other hand, the high-level structures for ABAC models in the current literature lack the expressiveness and uniformity often found in formal policy languages. We say that an ABAC model is *uniform* if the representations for different types of actions and their corresponding governing policies are in a consistent syntax. For instance, [4] does not support such uniformity as it specifies three types of actions for which their governing policies also have different representations. One type of action is for administrators creating and deleting a user, governed by fixed policies that are implicitly encoded in the model. Another type of action is for non-administrative users creating and modifying a subject and for their subjects creating and modifying an object, which are governed by various restricted forms of "constraints". The third type consists of other domain-dependent actions, such as read and write, governed by authorization predicates. Such non-uniform representation limits the expressiveness and usability of the model.

Previously, we have proposed to bridge the gap between the two research directions by combining the formalism of the high-level structure of an ABAC model with a policy language specification [14]. In RBAC literature, the combination of a high-level model and a policy language has been widely utilized. For instance, in [15], hierarchies are used to structure users, roles and groups and a simple policy language can be configured to support various conflict resolution mechanisms and propagation policies.

In this paper, we formalize such an ABAC model that uses a simple variant of description logics, $\mathcal{ALCN}_1$, to set out the high-level structure of how the components are represented and interact with each other, and that supports an expressive embedded policy language with a uniform treatment of actions using function-free first-order logic with equality. As described in [16], using logical formalisms would render clean foundations (hence formal guarantees), flexibility, and expressiveness. Moreover, description logics offer a natural representation of formalizing ontologies and have been used in various applications, such as diagnostic and configuration systems. Our proposed formalization would have the advantages of the current policy language specifications and of the current formalisms of high-level structures of models.

Our formalism would also be flexible in that it could be adjusted to incorporate additional desirable features, such as when time is a factor in making policy decisions regarding a requested action. In the many applications in which DAC, MAC, and RBAC are sufficient, the time of when an action is requested does not play a role. While in more complicated scenarios, such as large corporations and hospitals, whether an action can be authorized often depends on the time of the request. For example, a senior care facility might authorize its manager to delete the medical records of a resident only after at least seven years have passed since the resident left the facility. Offering such *flexibility* by allowing the designers of an ABAC system to tailor the functionalities of the underlying ABAC model to their requirements is a desirable feature of an ABAC formalism. The formalism in [9] shows how to incorporate conflict resolution and default policies. Our formalism offers a more generic framework by incorporating a variety of widely used features, including the time component, common mathematical domains, conflict resolution mechanisms, and default and exception policies.

The rest of the paper is organized as follows. Section II reviews the current literature on ABAC models. Section III defines the syntax and semantics of the description logic $\mathcal{ALCN}_1$ and function-free first-order logic with equality. Section IV discusses the detail of our ABAC model. Section VI describes some variations of our model to incorporate the extensions of time, conflict resolution mechanisms, and default and exception policies. Section VII introduces some techniques to achieve separation of duties. Section VIII concludes the paper.

II. Related Work

In this section we discuss current work on ABAC policies and models. We catalog some issues in current work and address those issues in the rest of the paper.

A. Policy Languages

Many formal authorization policy languages have been proposed, as surveyed in [16]. XACML [17] and PTaCL [7] are specifically designed for ABAC policies. XACML distinguishes the notions of policy sets, policies, and rules. A policy set may contain policies or other policy sets. A policy consists of one or more rules and the rules may be evaluated in a certain order. The hierarchy of policies reflect some real world requirements and is useful for defining default and exception policies, and for handling conflicts. A decision is made upon analyzing the subject's, object's, action's and environment's attributes, which are called the target of a policy set, policy, or rule. XACML is extremely expressive. However, it does not pose a generic structure or restrictions on the subjects, objects, actions and their attributes. Moreover, several semantics have been proposed for XACML [5], [18], [19] and an agreement on the standard has not been established yet.

The policy language PTaCL [7] provides the same problem space as XACML but with a formal semantics and simpler syntax. The semantics is defined by a three-valued logic. PTaCL is able to indicate whether some attribute values are missing. [20] extends PTaCL by implementing a probabilistic policy evaluation mechanism that probabilistically retrieves the missing attributes. Just as in case of XACML, these languages do not provide a formalization of the high-level structure of ABAC models.

Many other AC formal policy languages have been developed. For instance, the language PBel proposed in [6] is based on a four-valued logic, namely the Belnap logic. The work in [21] also proposes a generic framework for ABAC, but focuses on defining a policy language such that the resulting model is monotonic and complete instead of discussing in detail how the components are represented and interact with each other in the framework.

B. ABAC Models

This subsection reviews the current AC literature that focuses on developing high-level ABAC models, and explain in detail some of their common problems, especially the ones found in the logical models that use a symbol-set notation [3], [4], [22].

1) Semantics: The ABAC$_\alpha$ model [3], [4] representations are based on sets of tokens or symbols that denote users, subjects, objects, attributes, roles, and actions. ABAC$_\alpha$ regards attributes as functions whose domains and ranges are also sets of symbols. Intuitively, one regards a user attribute symbol $attr$ as a function on the set of users, one that maps each user u to the value $attr(u)$ of the attribute exhibited by the user. Each ABAC$_\alpha$ policy consists of a condition expressed in first-order syntax together with predicates from set theory, such as "is an element of" ($\in$) and "is a subset of" ($\subseteq$). For example, in the ABAC$_\alpha$ representation of DAC [3, Table 6, p. 51], the condition given for authorization of a subject s to read an object o is $SubCreator(s) \in reader(o)$, meaning that the user who created s must be one of the users given in the set of allowed readers of o.

Unfortunately, interpreting an attribute symbol as standing for a mapping from symbols to other symbols or sets of symbols divorces attributes from their intuitive meaning. Consider, for example, a user role indicating the organizational positions held by the user. In a symbol-set representation, this might mean that the value of Alice's orgPos attribute is a set $\{\mathit{StatutoryEmployee}, \mathit{Manager}, \dots\}$. What does $\mathit{Manager} \in \mathit{orgPos}(\mathit{Alice})$ mean? According to the definitions of ABAC$_\alpha$, it means that one symbol is in a set of symbols. This has little to do with the intended meaning, namely that Alice is a manager, or in standard logical notation, $\mathit{Manager}(\mathit{Alice})$. If one wants to say that managers can access all files owned by their subordinates, a direct statement would use predicate symbols to state properties of things. Stating this in terms of sets of symbols is an indirect expression that still leaves symbols such as $\mathit{Manager}$ without a formal interpretation.

It is important to note that mere use of logic in some way would not necessarily address the semantical issue. Indeed, the symbol-set view of ABAC$_\alpha$ is visible in several ABAC efforts that employ logical expressions [9], [23], [24]. For example, Wang et al. [9] present a logic-programming formalization of ABAC that uses predicates for sets, elements, and subsets, as in ABAC$_\alpha$, but that uses a different vocabulary to specify authorization conditions. Although the formalization in [9] is proved consistent and complete, it reasons about facts that describe its handling of sets of symbols, not about facts that describe entities in the ABAC system. Similarly, Finin et al. [23] recast RBAC using OWL [25], and Sharma and Joshi [24] recast ABAC$_\alpha$ using OWL and N3 [26]. Although these treatments do place AC entities within taxonomic hierarchies, these translations do not fully utilize the power of description logics and at their core depend on concepts that correspond directly to sets of attribute symbols, much as in ABAC$_\alpha$ [4], thus abandoning a good bit of the value of a logical formalization.

2) Readability and Maintainability: One hallmark of a convenient representation is that one can use it to describe the key things of interest in a simple and direct way, and that when time comes to modify the description, one can describe simple changes simply. Unfortunately, symbol-set representations fall short of this ideal with respect to simple modifications of a model. To extend a symbol-set model such as ABAC$_\alpha$ with new attributes, one must change the elements of one or more sets. If sets are defined by explicitly listing their elements, say by using $U = \{\mathit{Alice}, \mathit{Bob}, \mathit{Carol}\}$ to define the set of users and $\mathit{UA} = \{a_1, a_2, \dots, a_n\}$ to define the set of user attributes, adding a new user requires replacing the former list with a new one, say $U = \{\mathit{Alice}, \mathit{Bob}, \mathit{Carol}, \mathit{David}\}$, and making corresponding changes in other lists, such as the value of each of the attributes in UA. Modifying numerous lists offers numerous chances to make an error of omission, which one can expect to become more frequent as protected systems and organizations become larger and larger.

As shown in [16], one step toward minimizing the potential for error is to employ a more modular representation with incremental modification such that one just *adds* the new information, say by stating that David is a user, along with the values of David's attributes. Such modularity of statement lies at the heart of standard logical languages, and carries over to description logics as well. One also can make specification of non-additive changes, such as changing David's address, modular as well, as is common in databases.

3) Prohibitions: In many ABAC systems, authorization decisions are made with a bias toward denial, meaning that a request is authorized only if the antecedent of some authorization rule evaluates to true. This approach can cause problems when the system lacks the information needed to determine whether a condition is true or false. In other cases, the default might be that everyone has access apart from some exceptions. For example, all medical personnel in a hospital might have blanket permission to see any patient's medical records, except for the records of other medical personnel at the same hospital. Moreover, [27] has shown that having prohibitions along with authorization policies would reduce the total number of rules, and hence, the time for evaluating access requests.

The formal policy languages mentioned in Section II-A support both authorizations and prohibitions, some of which may even have explicit *undecided* decisions. But the ABAC models that focus on formalizing a high-level structure [3], [4] often have less expressive and direct representations of policies and they only explicitly support authorizations. Direct specification of the actual policies governing access would benefit from using explicit prohibitions in tandem with explicit authorizations. Thus, a policy language that handles both prohibitions and authorization would be suitable for AC models.

4) Terminological Axioms as Policies: A few works use description logics to formalize AC models [12], [28]. However, in these models, terminological (TBox) axioms of description logics are used to represent policies, rather than to construct a high-level ontology of the components of a model. For instance, in [28], the policy that says "all friends can download some music" is formalized as the terminological axiom $\mathit{Friend} \sqsubseteq \exists \mathit{Download}.\mathit{Music}$.

Even though terminological axioms are "syntactic sugar" of certain first-order formulas, treating policies as terminological axioms can be confusing and does not fully utilize the power of description logics in constructing a readable and maintainable ontology. For instance, the axioms $\mathit{Manager} \sqsubseteq \mathit{User}$ and $\mathit{Intern} \sqsubseteq \mathit{User}$ for specifying managers and interns as two types of users are obviously not policies. However, they are treated similarly as policies in a description logic knowledge base.

III. Background

In this section, we define the syntax and semantics of a variant of description logic and the function-free first-order logic with equality. Readers who are familiar with these logics may skip this section.

A. Description Logic $\mathcal{ALCN}_1$

A description logic (DL) provides a succinct syntax to express the ontology of a domain, specifically, what kinds of individuals are in the domain and what kinds of relationships the individuals have. Usually, a DL knowledge base consists of a *TBox* and an *ABox*. A TBox is to axiomatize the ontology of the domain using *concepts* and *roles*, where DL "roles" are a different notion than RBAC "roles". For instance, when formalizing a family tree, we might have concepts $People$ and $Female$, and a role $HasDaughter$ of $People$ with *fillers* in $Female$. An ABox assigns individuals to appropriate concepts and roles, e.g., $Female(x)$ means x is a female and $HasDaughter(x, y)$ means x has daughter y.

What differs a DL from another is the possible constructors on concepts and roles. We here define the minimal DL we use, $\mathcal{ALCN}_1$, to formalize our ABAC model. In $\mathcal{ALCN}_1$, a *concept expression* is defined inductively as

$$\begin{aligned} C ::= & A \mid \top \mid \bot \mid \neg C \mid C \sqcap C \mid C \sqcup C \mid {\leq}1R \\ & \mid {\geq}1R \mid \forall R.C \mid \exists R.C \end{aligned}$$

where A is an atomic concepts, and R is an atomic role. We only have atomic roles for simplicity. Note that since we have the negation of an arbitrary concept $\neg C$, the concepts $C \sqcup D$ and $\exists R.C$ are derivable from $C \sqcap D$ and $\forall R.C$, respectively, and vice versa. In this paper, we use capitalized words for atomic concept names, lower case words for atomic role names, named individuals and variables.

A TBox consists of general inclusion axioms of the form $C \sqsubseteq D$, meaning C is subsumed by D, where C and D are certain forms of concept expressions, discussed in detail in Section IV. In an ABox, an axiom is either a *concept assertion* of the form $C(a)$, or a *role assertion* of the form $R(a, b)$, meaning a is in concept C or a is related to b by role R respectively, where a and b are named individuals.

The semantics of $\mathcal{ALCN}_1$ is an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set, and $\cdot^{\mathcal{I}}$ is a function that maps a concept to a subset of $\Delta^{\mathcal{I}}$ and a role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The concept constructors are interpreted inductively as follows, where a and b are variables for the individuals.

$$\begin{array}{cc} A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} & \top^{\mathcal{I}} = \Delta^{I} \\ \bot^{\mathcal{I}} = \emptyset & (\neg C)^{\mathcal{I}} = \Delta^{I}/C^{\mathcal{I}} \\ (C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}} & (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}} \\ \multicolumn{2}{c}{({\leq}1R)^{\mathcal{I}} = \left\{a \in \Delta^{I} \,\middle|\, |\{b \mid (a,b) \in R^{\mathcal{I}}\}| {\leq} 1\right\}} \\ \multicolumn{2}{c}{({\geq}1R)^{\mathcal{I}} = \left\{a \in \Delta^{I} \,\middle|\, |\{b \mid (a,b) \in R^{\mathcal{I}}\}| {\geq} 1\right\}} \\ \multicolumn{2}{c}{(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{I} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}} \\ \multicolumn{2}{c}{(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{I} \mid \exists b.(a,b) \in R^{\mathcal{I}} \land b \in C^{\mathcal{I}}\}} \end{array}$$

An interpretation $\mathcal{I}$ satisfies a TBox axiom $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and we say $\mathcal{I}$ is a model of $C \sqsubseteq D$. Similarly, $\mathcal{I}$ satisfies an ABox axiom $C(a)$ if $a \in C^{\mathcal{I}}$, and $R(a, b)$ if $(a, b) \in R^{\mathcal{I}}$.

We introduce the following kinds of TBox axioms that are useful in formalizing ABAC. The first is to specify the domain and range of a role. Given a role R of concept C with fillers in concept D, C is R's *domain* and D is the *range*. Then we have axioms $\exists R.\top \sqsubseteq C$, and $\top \sqsubseteq \forall R.D$. The second kind of TBox axioms we will be using is to indicate that a role is *functional*, i.e., for every individual in the domain, exactly one individual in the range fills the role. Formally, given a functional role R with domain C and range D, we have TBox axioms $C \sqsubseteq {\leq}1R.D$, and $C \sqsubseteq {\geq}1R.D$. Third, to express two concepts C and D are disjoint from each other, we add an axiom $C \sqcap D \sqsubseteq \bot$.

B. Function-free First-order Logic with Equality

We use the function-free first-order logic with equality (FFOLE) to formalize the policies. A *term* t in FFOLE is used to refer to an individual, which is either a variable x or a constant c. An *atom* A is either a predicate or an equation between two terms, defined using the following syntactical rule:

$$A ::= P(t_1, ..., t_n) \mid t_1 = t_2.$$

A *literal* L is either an atom A or the negation $\neg A$ of atom A. Finally, a *formula* is inductively defined as follows:

$$\phi ::= L \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \rightarrow \phi \mid \forall x.\phi.$$

The semantics for FFOLE is an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set of named individuals, and $\cdot^{\mathcal{I}}$ is a function that maps each term to a named individual in $\Delta^{\mathcal{I}}$, and a predicate name P with arity n to a subset of $(\Delta^{\mathcal{I}})^n$. For instance, if $(a_1, a_2) \in P^{\mathcal{I}}$ where P is a predicate name with arity 2, then $P(a_1, a_2)$ evaluates to *true*.

IV. Model Formalization and Policy Specification

The components in a standard access control (AC) model are users, subjects, objects, actions, and policies. In this paper, what we mean by *users* are user accounts, instead of the human beings who own the accounts. *Subjects* are computer processes or login sessions of users. *Objects* are the accessible resources, which can be concrete such as a printer, or abstract such as a file in a computer. *Actions* generally are restricted to access control actions, such as reading and writing. *Policies* are rules used to decide whether an action is authorized or not.

AC models differ on what properties of the components can be used in expressing policies. In DAC, the properties are restricted to the owner property of the objects. In MAC, the properties are the security levels of the subjects and objects. In RBAC, the properties are restricted to the roles of the users, subjects and objects. In ABAC, the properties are generalized to include any user defined attributes.

The structure of our ABAC model is illustrated in Figure 1. An ABAC model is composed of two major parts: the ontology and policies. The *ontology* formalizes the high-level structure of a model, including the definitions of the components in the model and the relations among the components. We formalize the ontology using the description logic $\mathcal{ALCN}_1$ defined in Section III-A. The *policies* include the authorization policies that govern the actions in the model, each of which is represented as a formula in the function-free first-order logic with equality (FFOLE) defined in Section III-B.

In the ontology part of an ABAC model in Figure 1, each rounded box represents a component in the model, formalized as a concept in $\mathcal{ALCN}_1$, and each arrow represents an attribute, formalized as a role. An ABAC model consists of all the standard components in any AC model, namely policies, $User$, $Subject$, $Object$, and $Action$, and other domain-dependent types of entities, indicated by the boxes labeled $\langle other \rangle$. The concept $ActionObject$ subsumes all the concepts

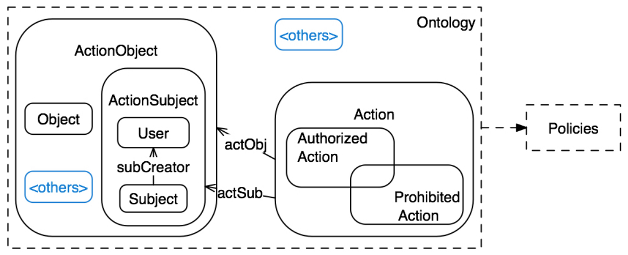

Fig. 1. Structure of an ABAC Model

that an action can act upon, and *ActionSubject* subsumes all the concepts that can perform an action. Follow the tradition of AC models, we restrict the subconcepts of *ActionSubject* to be only *User* and *Subject*, where the user actions are for logins and logouts. *AuthorizedAction* and *ProhibitedAction* are subconcepts of *Action* for authorized and prohibited actions, respectively. The attributes can be defined for any component in the ontology, including the domain-dependent ones. There are three distinguished attributes *subCreator*, *actSub*, *actObj*, indicating the creating user of a subject, the subject of an action, and the object of an action, respectively.

Formally, our ABAC model is defined as follows.

Definition 1 (ABAC Model): An *attribute-based access control (ABAC) model* is a tuple $\mathcal{M} = \langle \mathcal{C}, \mathcal{R}, \mathcal{T}, \mathcal{P} \rangle$, where

- $\mathcal{C}$ is a finite set of atomic concept names, including but not limited to *User*, *Subject*, *Object*, *Action*, *ActionObject*, *ActionSubject*, *AuthorizedAction*, and *ProhibitedAction* such that
  - *User*, *Subject* and *Object* are disjoint from each other and are subconcepts of *ActionObject*;
  - *Action* is disjoint from *ActionObject* and *ActionSubject*;
  - *AuthorizedAction* and *ProhibitedAction* are subconcepts of *Action*; and
  - *ActionSubject* is subsumed by $User \sqcup Subject$;
- $\mathcal{R}$ is a finite set of atomic role names, including but not limited to
  - *subCreator* with domain *Subject* and range *User*,
  - *actSub* with domain *Action* and range *ActionSubject*,
  - *actObj* with domain *Action* and range *ActionObject*;

  such that
  - *subCreator* and *actSub* are functional, and
  - *actObj* has maximum cardinality 1;
- $\mathcal{T}$ is a finite set of inclusion axioms, each of which is of one of the following forms:
  1) $C \sqsubseteq D_1 \sqcup D_2 \sqcup \cdots \sqcup D_n (n \geq 1)$, meaning C is subsumed by the union of $D_1, D_2, \ldots, D_n$;
  2) $C_1 \sqcap C_2 \sqsubseteq \bot$, meaning C_1 and C_2 are disjoint;
  3) $\exists R.\top \sqsubseteq C$, meaning the domain of R is C;
  4) $\top \sqsubseteq \forall R.D$, meaning the range of R is D;
  5) $C \sqsubseteq \leq 1 R.D$, and $C \sqsubseteq \geq 1 R.D$, meaning each individual in C is related to at most, and at least one individual in D by role R, respectively;

  where $C, D, C_i, D_i \in \mathcal{C} (1 \leq i \leq n)$, and $R \in \mathcal{R}$;
- $\mathcal{P}$ is a finite set of conditional FFOLE formulas whose predicate names are from $\mathcal{C}$ and $\mathcal{R}$. □

The DL concepts *User*, *Subject*, *Object*, and *Action* represent the users, subjects, objects, and actions in an ABAC model. The DL roles represent the attributes. We will discuss the policy language in detail in Section IV-C. Intuitively, a policy is a conditional formula whose antecedent represents the conditions under which an action is authorized or prohibited. The set of axioms $\mathcal{T}$ is essentially a restricted $\mathcal{ALCN}_1$ TBox. We list the axioms in $\mathcal{T}$ for the required concepts and attributes in Table I for clarification.

Subsumption relation between the concepts in $\mathcal{C}$	
$User \sqsubseteq ActionObject$	$Subject \sqsubseteq ActionObject$
$Object \sqsubseteq ActionObject$	$User \sqsubseteq ActionSubject$
$Subject \sqsubseteq ActionSubject$	$AuthorizedAction \sqsubseteq Action$
$ProhibitedAction \sqsubseteq Action$	$ActionSubject \sqsubseteq User \sqcup Subject$
Disjointness relation between the concepts in $\mathcal{C}$	
$User \sqcap Subject \sqsubseteq \bot$	$User \sqcap Object \sqsubseteq \bot$
$Subject \sqcap Object \sqsubseteq \bot$	$ActionObject \sqcap Action \sqsubseteq \bot$
$ActionSubject \sqcap Action \sqsubseteq \bot$	
Domain and range of each role in $\mathcal{R}$	
$\exists subCreator.\top \sqsubseteq Subject$	$\top \sqsubseteq \forall subCreator.User$
$\exists actSub.\top \sqsubseteq Action$	$\top \sqsubseteq \forall actSub.ActionSubject$
$\exists actObj.\top \sqsubseteq Action$	$\top \sqsubseteq \forall actObj.ActionObject$
Number restrictions of roles in $\mathcal{R}$	
$Subject \sqsubseteq \leq 1 subCreator.User$	
$Subject \sqsubseteq \geq 1 subCreator.User$	
$Action \sqsubseteq \leq 1 actSub.ActionSubject$	
$Action \sqsubseteq \geq 1 actSub.ActionSubject$	
$Action \sqsubseteq \leq 1 actObj.ActionObject$	

TABLE I
AXIOMS FOR REQUIRED CONCEPTS AND ROLES IN AN ABAC MODEL

The rest of this section explains the definition of our ABAC model in detail by drawing on the following case study of access control in an aged care facility's health information system reported in [29]. The ontology of the scenario is illustrated in Figure 2 using Graphol [30], a graphical language for drawing ontologies.

Example 1 (An Aged Care Facility): An aged care facility offers accommodation for some residents. It has a manager, who is in charge of the administrative duties including adding and deleting user accounts for the residents and staff. The health care workers at the facility are responsible for taking care of the residents. Doctors from nearby hospitals visit regularly and each of them is in charge of some residents. The Health Information System for the care facility stores the personal information, medical records, and medical insurance information of the residents. Medical records includes the past medical recodes, the care plan, and the private notes entered by the visiting doctors for each resident. □

A. Concepts

We define the *base concepts* in an ABAC model as *User*, *Subject*, *Object*, *Action*, *ActionObject*, *ActionSubject*,

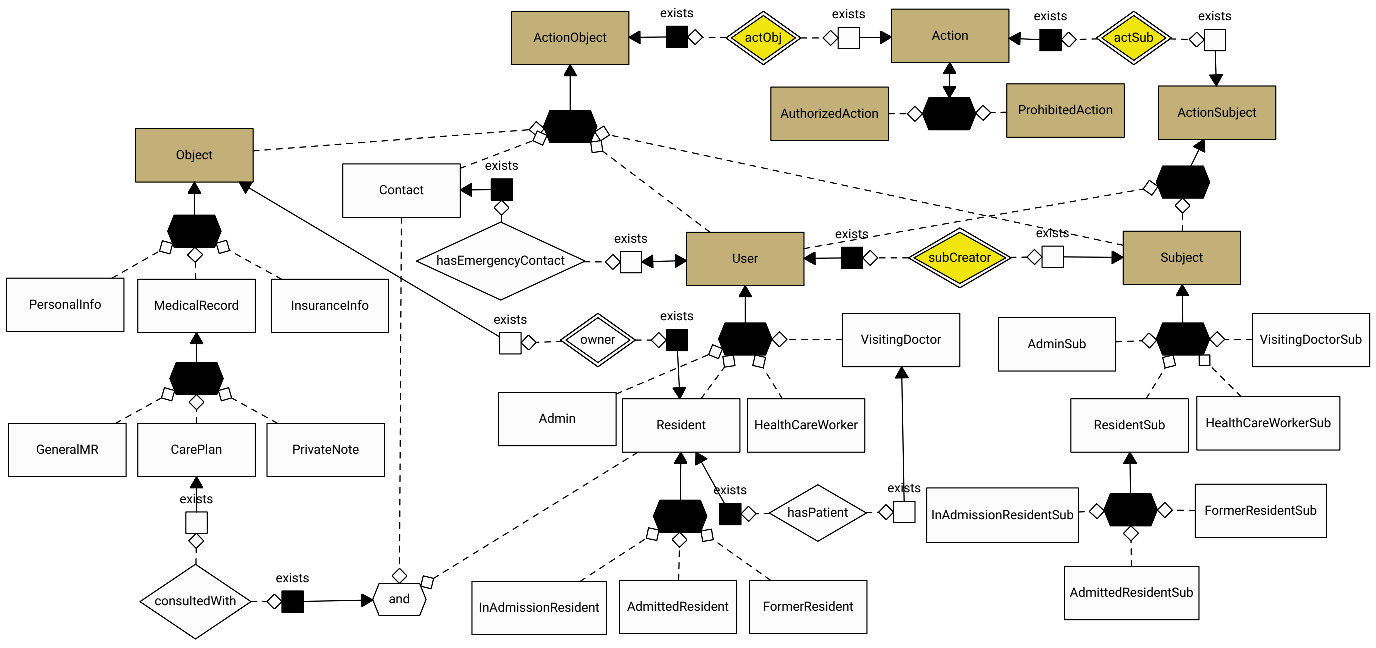

Fig. 2. Depiction of the ontology for the care facility scenario in Graphol[30]

AuthorizedAction, and *ProhibitedAction*. As mentioned earlier, the users in our model refer to user accounts rather than the human beings who own the accounts. The subjects are the login sessions of users, and the objects are resources in the system. Therefore, *User*, *Subject* and *Object* are disjoint from each other. We also require the concept *Action* to be disjoint from *ActionObject* for security reasons, e.g., a requested action would not be modified on the fly or an authorized action would not be modified to access other objects. The concepts *AuthorizedAction* and *ProhibitedAction* are subconcepts of *Action*, used for authorized and prohibited actions, respectively. Note that the two subconcepts of *Action* are not necessarily disjoint from each other, nor do they cover *Action*. Thus, an action may be both authorized and prohibited, or neither authorized nor prohibited.

1) Uniformity: To obtain a uniform representation of actions, we group the entities that an action may act upon into concept *ActionObject*, and that may perform an action into concept *ActionSubject*. The subconcepts of *ActionObject* are *User*, *Subject*, *Object*, and other domain-dependent concepts. For instance, a subject of an administrator user can create or delete a user, a subject or an object; a subject of a regular user can read or write an object. On the other hand, we require the only subconcepts of *ActionSubject* to be *User* and *Subject*. All actions, except for user logins and logouts, are performed by subjects. Different types of actions, e.g., read an object and create a user, are domain dependent and defined as subconcepts of *Action*. Moreover, for each concept whose entities can be created, we require a corresponding subconcept of *Action*. For example, if *User* has only two subconcepts U_1 and U_2 and all users can be created, then *Action* has a subconcept *CreateUserAction*, which has two further subconcepts $CreateU_1Action$ and $CreateU_2Action$. The reason is that when creating an individual a, since a does not exist yet, the policies cannot use it as an argument to specify which concept the to-be-created individual belongs to.

2) Maintainability: Most modifications of our model require only incremental changes to the set of axioms $\mathcal{T}$. To add a new subconcept U of *User*, one could simply add the axiom $U \sqsubseteq User$. If U is disjoint from another subconcept U' of *User*, one may add the axiom $U \sqcap U' \sqsubseteq \bot$. However, one may need to be careful with the usage of coverage axioms of the form $C \sqsubseteq D_1 \sqcup \cdots \sqcup D_n$ $(n > 1)$ because it might result in non-incremental modifications. Suppose an ABAC model contains an axiom $User \sqsubseteq U_1 \sqcup U_2$. To add another *User* subconcept U_3, the original axiom $User \sqsubseteq U_1 \sqcup U_2$ would needs to be modified to $User \sqsubseteq U_1 \sqcup U_2 \sqcup U_3$.

3) Concepts in Example 1: As shown in Figure 2, *User* has four disjoint subconcepts: *Admin*, *Resident*, *HealthCareWorker*, and *VisitingDoctor*. The concept *Resident* is further classified into the disjoint concepts *InAdmissionResident*, *AdmittedResident*, and *FormerResident*, for the residents who are in the admission process, who have been admitted and finished the admission process, and who have left the facility, respectively. The subsumption and disjointness axioms for the subconcepts of *User* are listed in Table II. We do not define any coverage axioms for maintainability, as discussed above.

The subconcepts of *Subject* are often defined similarly as those for *User*. In this care facility scenario, for each subconcept of *User*, there is a corresponding subconcept of *Subject*. For instance, the subconcept *AdminSub* of *Subject* corresponds to *Admin* and the *User* subconcept *ResidentSub* corresponds to *Resident*.

Subsumption	$Admin \sqsubseteq User$ $Resident \sqsubseteq User$ $HealthCareWorker \sqsubseteq User$ $VisitingDoctor \sqsubseteq User$ $InAdmissionResident \sqsubseteq Resident$ $AdmittedResident \sqsubseteq Resident$ $FormerResident \sqsubseteq Resident$
Disjointness	$Admin \sqcap Resident \sqsubseteq \bot$ $Admin \sqcap HealthCareWorker \sqsubseteq \bot$ $Admin \sqcap VisitingDoctor \sqsubseteq \bot$ $Resident \sqcap HealthCareWorker \sqsubseteq \bot$ $Resident \sqcap VisitingDoctor \sqsubseteq \bot$ $HealthCareWorker \sqcap VisitingDoctor \sqsubseteq \bot$ $InAdmissionResident \sqcap AdmittedResident \sqsubseteq \bot$ $InAdmissionResident \sqcap FormerResident \sqsubseteq \bot$ $AdmittedResident \sqcap FormerResident \sqsubseteq \bot$

TABLE II
AXIOMS FOR *User* IN THE CARE FACILITY SCENARIO

The classification of objects are often trickier than that of users due to the granularity of an object. In the care facility scenario, we have three subconcepts of $Object$: $PersonalInfo$, $MedicalRecord$ and $InsuranceInfo$, and $MedicalRecord$ is further classified into $GeneralMR$, $CarePlan$, and $PrivateNote$. For simplicity, we assume all of the personal information of a resident is contained in a single file, which is represented as one object in the information system, and the same holds for insurance information and care plans. As for the general medical records and private notes, the scenario is concerned about the multiple entries for a resident rather than the information as a whole. Thus we formalize each such entry is an object in concept $GeneralMR$ or $PrivateNote$.

The concept $Contact$ contains the individuals who are listed as the emergency contacts of some resident. The administrators are authorized to add or delete an individual from $Contact$, thus it is a subconcept of $ActionObject$.

Figure 3 shows the (partial) hierarchy of $Action$ in the scenario. $Action$ has four immediate subconcepts, $ReadAction$, $WriteAction$, $CreateAction$, and $DeleteAction$. As mentioned earlier, $CreateAction$ needs to be further classified according to the kinds of individuals that can be created. Besides the subconcepts $CreateContactAction$, $CreateUserAction$, $CreateSubAction$ and $CreateObjAction$ in the figure, there are also concepts $CreateResidentAction$ and $CreateAdminAction$, etc.. We also require that for each subconcept of $Action$, its subconcepts are disjoint from each other. For instance, an action cannot both delete an object and create a user account.

B. Attributes

The *base attributes* in an ABAC model are $subCreator$, $actSub$, and $actObj$. The attribute $subCreator$ indicates the creating user of a subject and it is functional, i.e., each subject has exactly one creating user. Every time a user logs in, a subject is created. The attribute $actSub$ indicates the subject, or the performer, of an action and it is also functional. We do not consider collaborative actions in this paper. The attribute $actObj$ indicates the object of an action. However, $actObj$

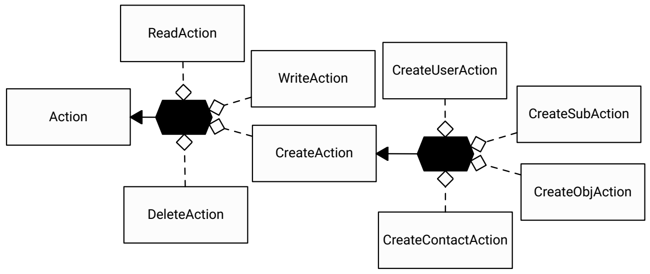

Fig. 3. Types of actions in the care facility scenario

has max cardinality 1 instead of being functional because the policies governing the actions for creating an individual cannot take the to-be-created individual as an argument.

In cases where the to-be-created individual is needed in some attribute to express a policy, we define a corresponding attribute for the action such that the action takes the place of the to-be-created individual. The definition for how to specify such an action attribute is in Definition 2. For instance, suppose an attribute $owner$ of an object o is needed in a policy for creating the object, we define an attribute $ownerActSpec_1$ for the creation action a with the same value u as $owner$. Thus, instead of $owner(o, u)$, we use $ownerActSpec_1(a, u)$ in the policy definition.

Definition 2 (Action Specification Attribute): Let $attr$ be an attribute on x with value y, i.e., $attr(x, y)$. If a is an action that creates the individual x, then the *action specification attribute* for $attr$ on x is an attribute $attrActSpec_1$ on a with value y, i.e., $attrActSpec_1(a, y)$. Similarly, if a is an action that creates the individual y, then the *action specification attribute* for $attr$ on y is an attribute $attrActSpec_2$ on x with value a, i.e., $attrActSpec_2(x, a)$.

1) Maintainability: Similar as the representation for concepts, the representation for attributes in our formalism is also modular and its modification is incremental. For instance, to add a new attribute $attr$ whose domain is C and whose range is D, one only needs to add the axioms $\exists attr.\top \sqsubseteq C$ and $\top \sqsubseteq \forall attr.D$. Similarly, cardinality and functionality constraints $C \sqsubseteq \leq 1 attr.D$ and $C \sqsubseteq \geq 1 attr.D$ can also be easily added without modifying existing axioms

2) Attributes in Example 1: Table III lists the attributes we use in formalizing some of the policies in the care facility scenario. The attribute $owner$ indicates the owner resident of an object and it is functional. The attribute $hasEmergencyContact$ indicates the emergency contact of a resident and each resident must have exactly one emergence contact. The $consultedWith$ attribute means a care plan has been consulted with a resident or a resident's emergency contact and its minimum cardinality restriction is one. Finally, $hasPatient$ indicates the residents each visiting doctor is responsible for.

C. Policies

A *policy* for an action is a conditional formula in FFOLE. We only consider authorization and prohibition policies in this

Attribute Name	Domain	Range
$owner$	$Object$	$Resident$
$hasEmergencyContact$	$Resident$	$Contact$
$consultedWith$	$CarePlan$	$Resident \sqcup Contact$
$hasPatient$	$VisitingDoctor$	$Resident$

TABLE III
SOME ATTRIBUTES IN THE AGED CARE FACILITY SCENARIO

paper. For each policy, the consequent is the requested action being either authorized or prohibited, and the antecedent is the preconditions need to be satisfied to authorize or prohibit the action. As mentioned earlier, the actions for creating an individual are essentially different from the other types of actions. Therefore, we formalize the policies governing creation actions in a slightly different way than the policies governing other types of action. The formal definitions for the two types of policies are in Definition 3 and Definition 4.

Definition 3 (Non-creation Policy): Let $\mathcal{M} = \langle \mathcal{C}, \mathcal{R}, \mathcal{T}, \mathcal{P} \rangle$ be an ABAC model defined in Definition 1. A *policy for a non-creation action* in $\mathcal{M}$ is a conditional formula in the following form:

$$[C_1(a) \wedge actSub(a,s) \wedge actObj(a,o) \wedge \\ C_2(s) \wedge C_3(o) \wedge \psi(a,s,o,\vec{x},\vec{c})] \rightarrow P(a),$$

where a, s, and o are free variables representing the requested action, the subject of the action, and the object of the action, respectively. C_1, C_2, and C_3 are arbitrary $\mathcal{ALCN}_1$ concepts constructed from concept names in $\mathcal{C}$ such that they are subconcepts of $Action$, $ActionSubject$, and $ActionObject$, respectively. And $\psi(a,s,o,\vec{x},\vec{c})$ is a formula in FFOLE specifying the rest of the preconditions, in which the only predicate names are concept names in $\mathcal{C}$ and role names in $\mathcal{R}$, $\vec{x}$ represents the free variables other than a, s, and o, and $\vec{c}$ is for the constants used ψ. The consequent $P(a)$ is either $AuthorizedAction(a)$ or $ProhibitedAction(a)$.

Definition 4 (Creation Policy): Let $\mathcal{M} = \langle \mathcal{C}, \mathcal{R}, \mathcal{T}, \mathcal{P} \rangle$ be an ABAC model defined in Definition 1. A *policy for a creation action* is in the following form:

$$[C_1(a) \wedge actSub(a,s) \wedge C_2(s) \wedge \phi(a,s,\vec{x},\vec{c})] \rightarrow P(a),$$

where the symbols are defined similarly as above in Definition 3. However, the binary predicates used here cannot directly involve the to-be-created individual. Instead, we use the corresponding action specification attribute (Definition 2) for the to-be-created individual. □

1) Uniformity: Besides the necessity to have two different representations for policies, one for creation actions and the other for non-creation ones, each of the two representations is able to express a wide range of policies. For instance, the policies for the subjects of regular users reading an object, and for those of administrator users modifying the attributes of a user can both be represented using non-creation policies. Similarly, the policies for the subjects of administrators creating a user and for user logins can also both be represented using creation policies.

2) Effects of policies: We do not define a language specification for the effects of policies in this paper. Intuitively, when an action is authorized, the set of assertions in the ABAC configuration $\mathcal{S}$ is updated accordingly. For example, if an action for adding the value v to an attribute $attr$ for an individual x is authorized, then the role assertion $attr(x,v)$ is added, assuming v is already defined. By Definition 5 (Section IV-D), an ABAC configuration is essentially an $\mathcal{ALCN}_1$ ABox knowledge base with a fixed set of individuals for interpretation. The details on how to update an ABox and the corresponding interpretation can be found in [31].

3) Policies in Example 1: In this subsection, we illustrate the usage of policies through some example policies in the care facility case study, one for each of the action types $ReadAction$, $WriteAction$, $DeleteAction$, $CreateSubAction$, and $CreateUserAction$.

Example 2: The health care workers can view the medical records of the residents. Formalized as:

$$[ReadAction(a) \wedge actSub(a,s) \wedge actObj(a,o) \wedge \\ HealthCareWorkerSub(s) \wedge MedicalRecord(o)] \rightarrow \\ AuthorizedAction(a).$$

Example 3: Only an administrator can update the care plan of a resident and the care plan is updated in consultation with the resident or the resident's emergency contact. Formalized as the following two policies. Note that due to the way we formalize the policies here, if an administrator requests to update the care plan of a resident without consulting with the resident or his/her emergency contact, we do not have a conclusive decision on whether the update is authorized or prohibited.

$$[WriteAction(a) \wedge actSub(a,s) \wedge actObj(a,o) \wedge \\ AdminSub(s) \wedge CarePlan(o) \wedge subCreator(s,u) \wedge \\ owner(o,u) \wedge (consultedWith(o,u) \vee \\ (hasEmergencyContact(u,p) \wedge consultedWith(o,p)))] \\ \rightarrow AuthorizedAction(a)$$

$$[WriteAction(a) \wedge actSub(a,s) \wedge actObj(a,o) \wedge \\ \neg AdminSub(s) \wedge CarePlan(o)] \rightarrow ProhibitedAction(a)$$

Example 4: Only the medical records of a resident who has left the facility can be deleted and they are to be deleted by an administrator. Formalized as:

$$[DeleteAction(a) \wedge actSub(a,s) \wedge actObj(a,o) \wedge \\ AdminSub(s) \wedge subCreator(s,u) \wedge owner(o,u) \wedge \\ FormerResident(u)] \rightarrow AuthorizedAction(a),$$

$$[DeleteAction(a) \wedge actSub(a,s) \wedge actObj(a,o) \wedge \\ (\neg AdminSub(s) \vee (subCreator(s,u) \wedge owner(o,u) \wedge \\ \neg FormerResident(u)))] \rightarrow ProhibitedAction(a).$$

Example 5: A visiting doctor can create private notes for the patients he/she is responsible for. Formalized as:

$$[\mathit{CreateCarePlanAction}(a) \wedge \mathit{actSub}(a,s) \wedge \mathit{VisitingDoctorSub}(s) \wedge \mathit{subCreator}(s,u) \wedge \mathit{ownerActSpec}_1(a,p) \wedge \mathit{hasPatient}(u,p)] \rightarrow \mathit{AuthorizedAction}(a),$$

$$[\mathit{CreateCarePlanAction}(a) \wedge \mathit{actSub}(a,s) \wedge \mathit{VisitingDoctorSub}(s) \wedge \mathit{subCreator}(s,u) \wedge \mathit{ownerActSpec}_1(a,p) \wedge \neg \mathit{hasPatient}(u,p)] \rightarrow \mathit{ProhibitedAction}(a).$$

Example 6: A user can only log in as a subject that corresponds to the concept the user is in. We need one policy for each subconcept of User. For instance, one for administrators Admin, and one for admitted residents $\mathit{AdmittedResident}$. The policy for an administrator user login is formalized as:

$$[\mathit{CreateAdminSubAction}(a) \wedge \mathit{actSub}(a,u) \wedge \mathit{Admin}(u)] \rightarrow \mathit{AuthorizedAction}(a).$$

D. ABAC Configuration

An ABAC model defines the generic structure of the components and policies in an application domain. In this paper, we refer to an *ABAC configuration* as a specification of an ABAC model that fills the components with named individuals. The formal definition for an ABAC configuration is in Definition 5.

Definition 5 (ABAC Configuration): Let $\mathcal{M} = \langle \mathcal{C}, \mathcal{R}, \mathcal{T}, \mathcal{P} \rangle$ be an ABAC model. An *ABAC configuration* of $\mathcal{M}$ is a tuple $\mathcal{S} = \langle \mathcal{M}, \Delta, \mathcal{A} \rangle$, where

- Δ is a finite set of named individuals;
- $\mathcal{A}$ is a finite set of axioms, each of which is either of the form $C(a)$ or of $R(a,b)$, where $C \in \mathcal{C}$, $R \in \mathcal{R}$, and $a, b \in \Delta$. □

Intuitively, an ABAC model $\mathcal{M}$ forms an $\mathcal{ALCN}_1$ TBox knowledge base with governing policies, and an ABAC configuration is an ABox knowledge base with a fixed set of individuals. Suppose Alice is a visiting doctor of the care facility and Bob is one of Alice's patients, then we have the individuals $\mathit{Alice}, \mathit{Bob} \in \Delta$, and axioms $\mathit{hasPatient}(\mathit{Alice}, \mathit{Bob})$, $\mathit{VisitingDoctor}(\mathit{Alice})$, $\mathit{Resident}(\mathit{Bob}) \in \mathcal{A}$.

Let $\mathcal{S} = \langle \mathcal{M}, \Delta, \mathcal{A} \rangle$ be an ABAC configuration of an ABAC model $\mathcal{M} = \langle \mathcal{C}, \mathcal{R}, \mathcal{T}, \mathcal{P} \rangle$. All of the information specified in $\mathcal{A}$ can be stored in a database, which can then be checked against the axioms in $\mathcal{T}$ to make sure the constraints are satisfied. Given a policy $\phi \in \mathcal{P}$, since all variables in ϕ are free, they are substituted for the named individuals in Δ, and whether $C(a)$ or $R(a,b)$ is true can be determined by a simple lookup of the tables in the database. Suppose $\mathcal{S}$ is consistent w.r.t. $\mathcal{M}$, then each unary predicate $C(x)$ in ϕ evaluates to *true* if $C(x) \in \mathcal{A}$, i.e., $C(x)$ is stored in the database. Otherwise, $C(x)$ is *false*. Similarly, each binary predicate $R(x,y)$ in ϕ evaluates to *true* if $R(x,y) \in \mathcal{A}$, and *false* otherwise. For instance, in Example 5, even if we do not have negation of roles in $\mathcal{ALCN}_1$, $\neg \mathit{hasPatient}(u,p)$ would evaluate to true if $\mathit{hasPatient}(u,p)$ is not in $\mathcal{A}$.

V. Implementation in OWL and N3

The formalism of our ABAC model can be easily implemented using OWL 2 [32] and N3 [26]. We briefly discuss the implementation in this section.

A component in the ontology of our ABAC model is represented as a concept in $\mathcal{ALCN}_1$, corresponding to a `class` in OWL 2. The base concepts can be defined as in the following.

```
User a owl:class.
Subject a owl:class.
Object a owl:class.
Action a owl:class.
AuthorizedAction a owl:class.
ProhibitedAction a owl:class.
ActionObject a owl:class.
ActionSubject a owl:class.
```

The subsumption axioms are listed as follows, using the `subClassOf` RDF schema.

```
User rdfs:subClassOf ActionObject.
Subject rdfs:subClassOf ActionObject.
Object rdfs:subClassOf ActionObject.
User rdfs:subClassOf ActionSubject.
Subject rdfs:subClassOf ActionSubject.
ActionSubject rdfs:subClassOf
    owl:unionOf (User Subject).
AuthorizedAction rdfs:subClassOf Action.
ProhibitedAction rdfs:subClassOf Action.
```

The axioms for the disjointness of two concepts can be enforced using the OWL statement `disjointWith`, listed as following:

```
User owl:disjointWith Subject.
User owl:disjointWith Object.
Subject owl:disjointWith Object.
ActionObject owl:disjointWith Action.
ActionSubject owl:disjointWith Action.
```

An attribute in an ABAC model is represented as an $\mathcal{ALCN}_1$ role, corresponding to an `ObjectProperty` in OWL 2. The axioms for the domain, range, and number restrictions of each role are listed as follows, using the OWL statements `FunctionalProperty`, `maxCardinality` and `minCardinality`, and RDF schemas `domain` and `range`.

```
subCreator a owl:ObjectProperty,
  owl:FunctionalProperty;
  rdfs:domain Subject;
  rdfs:range User.
actSub a owl:ObjectProperty,
  owl:FunctionalProperty;
  rdfs:domain Action;
  rdfs:range ActionSubject;
actObj a owl:ObjectProperty;
  owl:maxCardinality "1"^^xsd:nonNegativeInteger;
  rdfs:domain Action;
  rdfs:range ActionObject.
```

The property of modular representation and incremental modification of our ABAC formalism is also preserved in the OWL 2 implementation. To add a new concept, the corresponding subsumption and disjointness axioms can be added without modifying the existing axioms. Similarly, to add a new attribute, the axioms for its domain, range, and number restrictions can also be added without modifying the existing axioms.

We express policies using N3 syntax. In an N3 rule, each symbol `?X` starting with `?` is a free variable. An individual x is in concept C, i.e., $C(x)$ is written as `x a C` and $R(x, y)$ is written as `x R y`. For instance, the policy in Example 2 for health care workers viewing the medical records of residents is represented as an N3 rule in the following.

```
{?A a ReadAction;
    actSub ?S;
    actObj ?O.
 ?S a HealthCareWorkerSub.
 ?O a MedicalRecord.} => {?A a AuthorizedAction.}.
```

The policy for administrator user logins is expressed as the following N3 rule.

```
{?A a CreateAdminSubAction;
    actSub ?U.
 ?U a Admin.} => {?A a AuthorizedAction.}.
```

The axioms $\mathcal{A}$ in an ABAC configuration $\mathcal{S} = \langle \mathcal{M}, \Delta, \mathcal{A} \rangle$ can also be easily implemented in OWL 2. For instance, the fact that the resident *Bob* is a patient of the visiting doctor *Alice* is expressed as in the following. We use capitalized words for named individuals only in this section to keep consistent with OWL 2 syntax.

```
Alice a VisitingDoctor.
Bob a AdmittedResident.
Alice hasPatient Bob.
```

Given an OWL 2 implementation of the axioms and an N3 implementation of the policies, one may use the EYE reasoner to evaluate whether an action is authorized or prohibited [33].

VI. Incorporating Additional Features

The formalism of our ABAC model can be modified to incorporate a wide range of additional features. We discuss in this section how to add time, conflict resolution mechanisms, and default and exception policies.

A. Time

In many applications, the policies take time into consideration when making decisions, such as the time when an action is requested. To be able to talk about time in our formalism, we need a distinguished individual representing the environment or context and a way of representing the timestamps in the environment.

First, we add another concept constructor to $\mathcal{ALCN}_1$, namely the "one-of" constructor. $\{a_1, \ldots, a_n\}$. We formalize the environment as a named individual *environment* and create a concept *Env* that only contains *environment*, which can be expressed by adding the following axioms to the set of axioms $\mathcal{T}$ of an ABAC model $\mathcal{M}$ defined in Definition 1:

$$Env \sqsubseteq \{environment\}, \text{ and } \{environment\} \sqsubseteq Env.$$

Accordingly, the definition of an ABAC configuration $\mathcal{S} = \langle \mathcal{M}, \Delta, \mathcal{A} \rangle$ is modified by including the individual *environment* in Δ and the concept assertion $Env(environment)$ in $\mathcal{A}$. To keep the runtime complexity for checking whether the axioms in $\mathcal{T}$ are satisfied reasonably low, one may disallow the one-of constructor to be used in arbitrary concepts.

Adding timestamps is more complicated than adding the environment. One approach is to formalize a temporal version of $\mathcal{ALCN}_1$. In this paper, we discuss another simpler approach by adding *concrete domains* to our ABAC model [34]. A concrete domain $\mathcal{D}$ is simply a set of individuals $\Delta^{\mathcal{D}}$ that have a set of predicates $pred(\mathcal{D})$ other than the ones in a DL knowledge base defined upon them. For instance, the concrete domain $\mathcal{N}$ consists of the set of all natural numbers, the binary predicate names $<$, $\leq$, $>$, and $\geq$, and the unary predicate names $<_n$, $\leq_n$, $>_n$, and $\geq_n$, defined in the obvious way. These predicates are called *concrete predicates*. For simplicity, we use $\mathcal{N}$ to represent time in our model. The resulting description logic after adding *Env* and $\mathcal{N}$ to $\mathcal{ALCN}_1$ is still decidable [35].

Example 7: We define the functional attributes $currentTime$ for the current time of $environment$ and $leftTime$ for the time when a resident leaves the care facility. The domain of $currentTime$ is Env and range is the set of natural numbers. The domain of $leftTime$ is $FormerResident$ and range is the set of natural numbers. Suppose the natural numbers represent the years. Then the policy that says "an administrator can delete a resident's medical records if the resident left the facility before 2000 and it is after 2007 now" is formalized as follows, where $>$ and $<$ are concrete predicates in $\mathcal{N}$.

$$\begin{aligned}&[DeleteAction(a) \wedge actSub(a, s) \wedge actObj(a, o) \wedge \\ &AdminSub(s) \wedge subCreator(s, u) \wedge owner(o, u) \wedge \\ &FormerResident(u) \wedge currentTime(environment, t_1) \wedge \\ &(t_1 > 2007) \wedge leftTime(u, t_2) \wedge (t_2 < 2000)] \rightarrow \\ &AuthorizedAction(a)\end{aligned}$$

B. Conflict Resolution

Explicit conflicts among policies can arise if the policies can express both authorizations and prohibitions. Explicit conflicts cannot arise when using either a prohibition- or permission-by-default approach; instead, implicit conflicts between rules would manifest as erroneous or unexpected decisions, with the expected governing rule failing to produce an authorization, but an unexpected rule producing an authorization.

Space limitations preclude detailed discussion of resolving conflicting policies. We here only briefly demonstrate that our formalism is able to incorporate conflict resolution mechanisms rather than discuss the details of how the mechanisms work. One simple approach to incorporate conflict resolution in our formalism is to assign priorities to policies. When two policies are in conflict, the policy with the higher priority takes precedence. However, this approach is not maintainable. When a new policy is added, the priorities of the other policies may need to be adjusted as well. Another similar approach is to define a predicate $dominates(p_1, p_2)$ between two policies such that when p_1 and p_2 are in conflict, p_1 takes precedent over p_2. This approach would be more maintainable but it may result in cyclic dominating relations.

Our framework is also able to incorporate other various kinds of conflict resolution techniques, such as inheritance hierarchies with exceptions [36], answer set programming [37], and preference-based nonmonotonic or default argumentation techniques [36], [38], [39].

C. Default and Exception Policies

A default policy is useful when an action is not governed by any other policies defined in the system. Our formalism is able to incorporate this feature by simply adding policies of the following forms:

$$[C_1(a) \land \neg \mathit{AuthorizedAction}(a) \land \phi(a, \vec{x}, \vec{c})] \rightarrow \mathit{ProhibitedAction}(a)$$

$$[C_1(a) \land \neg \mathit{ProhibitedAction}(a) \land \phi(a, \vec{x}, \vec{c})] \rightarrow \mathit{AuthorizedAction}(a),$$

where a is a free variable representing a requested action, C_1 is a subconcept of Action, and $\phi(a, \vec{x}, \vec{c})$ represents the rest of the preconditions with free variables in $\vec{x}$ and constants in $\vec{c}$. In this way, our formalism allows a granular incorporation of default policies, as illustrated in the following Example. Different types of users may subject to different default policies for the same action. Similarly, different types of actions may be governed by different default policies.

Example 8: In the care facility scenario, a health care worker is authorized to read the medical records of any resident unless said otherwise, while a visiting doctor is prohibited to read the medical records unless said otherwise. Formalized as the following two policies, respectively.

$$[\mathit{ReadAction}(a) \land \neg \mathit{ProhibitedAction}(a) \land \mathit{actSub}(a, s) \land \mathit{HealthCareWorkerSub}(s)] \rightarrow \mathit{AuthorizedAction}(a)$$

$$[\mathit{ReadAction}(a) \land \neg \mathit{AuthorizedAction}(a) \land \mathit{actSub}(a, s) \land \mathit{VisitingDoctorSub}(s)] \rightarrow \mathit{ProhibitedAction}(a) \quad \square$$

Furthermore, the close and open policies can be formalized as in the following, respectively.

$$[\mathit{Action}(a) \land \neg \mathit{AuthorizedAction}(a)] \rightarrow \mathit{ProhibitedAction}(a)$$

$$[\mathit{Action}(a) \land \neg \mathit{ProhibitedAction}(a)] \rightarrow \mathit{AuthorizedAction}(a)$$

Adding exception policies often result in conflicting policies. We here only demonstrate our formalism's ability to add exception policies. Conflict resolution mechanisms can be further incorporated as discussed in Section VI-B.

To add an exception policy pol_1 for an original policy pol_2, one can simply assign a higher priority to pol_2 than pol_1. Thus, pol_2 would take precedence when authorizing a requested action. In this way, one would not need to modify the original policies.

Example 9: Suppose the environment is defined in the care facility scenario using the one-of constructor. Let the following be a policy originally defined in the model with priority 2 for prohibiting anyone who is not a health care worker from reading a resident's medical record.

$$[\mathit{ReadAction}(a) \land \mathit{actSub}(a, s) \land \mathit{actObj}(a, o) \land \neg \mathit{HealthCareWorkerSub}(s) \land \mathit{MedicalRecord}(o)] \rightarrow \mathit{ProhibitedAction}(a)$$

To add an exception that says any staff of the facility and visiting doctors can read any resident's medical records in case of an epidemic, one may add the following policy with priority 3, or any number greater than 2:

$$[\mathit{ReadAction}(a) \land \mathit{actSub}(a, s) \land \mathit{actObj}(a, o) \land (\mathit{HealthCareWorkerSub}(s) \lor \mathit{AdminSub}(s) \lor \mathit{VisitingDoctorSub}(s)) \land \mathit{MedicalRecord}(o) \land \mathit{InEmergency}(\mathit{environment}, \mathit{epidemic})] \rightarrow \mathit{AuthorizedAction}(a),$$

where $\mathit{InEmergency}(\mathit{environment}, \mathit{epidemic})$ indicates the facility is in epidemic. □

To avoid such potential conflict between an original and an exception policy, one could also modify the original policy such that it is only applicable when there is no ongoing epidemic, as expressed in the following.

$$[\mathit{ReadAction}(a) \land \mathit{actSub}(a, s) \land \mathit{actObj}(a, o) \land \neg \mathit{HealthCareWorkerSub}(s) \land \mathit{MedicalRecord}(o)] \rightarrow \neg \mathit{InEmergency}(\mathit{environment}, \mathit{epidemic})] \rightarrow \mathit{ProhibitedAction}(a)$$

However, when there are many exceptions, modifying the original policy to take into account the exception conditions may result in policies that are too complicated to understand and maintain.

VII. Separation of Duties

Separation of duties (SoD) is a fundamental principle in security systems, requiring that more than one users are needed to perform a critical task so that no single user is able to control all the steps involved in the task. A generalization of the principle is a k-n SoD, stating that given a task composed of n permissions, no less than k $(1 < k \leq n)$ users are required to perform the task. A *permission* is simply a pair of an action and an object. The n permissions do not necessarily involve different objects or types of actions. Jha et.al. [40], [41] analyze the k-n SoD problem in an ABAC model using a symbol-set notation and propose a method for implementing a limited version of SoD using mutually exclusive policies, specifically when $k = n$. The rest of this section discusses how to achieve limited versions of k-n SoD using mutual exclusion of concepts and attributes, or using more expressive policies.

A. Mutual Exclusion

When $k = n$ ($n > 1$), a k-n SoD constraint simply states that given n permissions, each of them needs to performed by a different user. For example, in the control of implementing an information system, the person who implements the system cannot be the same person who reviews and tests it.

One may use a similar approach as in RBAC to achieve such SoD constraint, i.e., through the usage of mutually exclusive, or disjoint, concepts. To achieve static separation of duties (SSoD), one may specify that some subconcepts of $User$ are disjoint. For instance, suppose we have two $User$ subconcepts: $Developer$ for developing an information system, and $Tester$ for reviewing and testing the system. To satisfy the requirement that says no single user can be both a developer and a tester, one may simply add the disjoint axiom $Developer \sqcap Tester \sqsubseteq \bot$. On the other hand, to achieve dynamic separation of duties (DSoD), one may require some subconcepts of $Subject$ to be disjoint. For instance, a user can be both a developer and a tester, but it cannot login as a developer and a tester at the same time, i.e., $DeveloperSub \sqcap TesterSub \sqsubseteq \bot$.

However, the above approach for achieving DSoD using disjoint subconcepts of $Subject$ is not sufficient or appropriate in some cases. In the information system development example, the subconcepts $Developer$ and $Tester$ of $User$ may be required to overlap in some cases. A user who is both a developer and a tester would breach the security by logging in first as a developer to implement a system and then logging in as a tester to test the same system. In such cases, our formalism needs to be extended to be able to state that the two functionalities of developing and testing are disjoint given the same object. Such disjointness can be achieved by adding disjoint axioms between $\mathcal{ALCN}_1$ roles. For example, suppose each information system that is under development is an object, and it has two attributes: $developedBy$ and $testedBy$, with range $User$. The DSoD requirement can be simply stated as $disjoint(developedBy, testedBy)$, meaning for any object, no single user can both develop and test it.

B. k-n SoD

A k-n SoD requirement states that suppose a task t is composed of n ($n > 1$) permissions, each of which is pair (a, o) where a is an action and o is an object, then at least k ($1 < k \leq n$) users are needed to complete task t. Whether a k-n SoD requirement is static or dynamic depends on the time it is applied.

Suppose the n actions in the permissions are to be performed sequentially. We illustrate how to (partially) enforce a k-n SSoD for the sequential actions through the following example. A k-n DSoD can be specified similarly. The idea is that for each type of action, we define an $ActionObject$ functional attribute to indicate the creating user of a subject that performs an action of the type on the object. For instance, if the subject s with creating user u places an order o, we have $orderedBy(o, u)$. Then the last action in the n permissions is authorized if there are at least k distinct users among the $n-1$ (not necessarily distinct) users who have completed the first $n-1$ actions and the user who requests the last action.

Example 10: An inventory task is composed of ordering goods from suppliers, receiving goods from suppliers, and logging in the received goods, and at least 2 (at most 3) users are required to perform the task. We represent each order as an object, and the inventory task as three actions: $OrderAction$, $ReceiveOrderAction$, and $LoginOrderAction$. After an oder is placed and received, the functional attributes $orderedBy$ and $receivedBy$ for the order is updated accordingly to indicate which user(s) placed the received the order. Then the policy for logging in an order is formalized as:

$$\begin{gathered}[LoginOrderAction(a_3) \wedge actSub(a_3, s) \wedge actObj(a_3, o) \wedge \\ subCreator(s, u_3) \wedge orderedBy(o, u_1) \wedge receivedBy(o, u_2) \\ \wedge ((u_3 \neq n_2) \vee (u_3 \neq n_1))] \rightarrow AuthorizedAction(a)\end{gathered}$$

Note that in the above example, all 3 of the actions of the task act on the same object and $k = 2$. In more complicated cases where more than one objects are involved or $k > 2$, the policy language needs to be extended to include existential quantifiers. However, this approach does not prevent potential violations of a k-n SoD by prohibiting any of the first $n-1$ actions.

VIII. Conclusion and Future Work

This paper proposes a direct logical formalism of ABAC models using a variant of description logics and function-free first-order logic with equality. Our formalism has a clear and well-defined semantics for both the high-level structure of the model and the language specification. The model also has a uniform representation for different types of actions and their governing policies, such as creating an individual and reading an object.

We are able to demonstrate that our formalism is more maintainable for administrators than the set- or tuple-based formalisms. Adding a new concept of individuals is easily done by adding extra axioms to the knowledge base to indicate the added concept's superconcept and its disjoint concepts. Adding a new attribute is also easily done by adding extra axioms to indicate its domain, range, and cardinality constraints. We demonstrate the flexibility of our formalism by illustrating how to add several widely used features, including the environment, time, conflict resolution mechanisms, and default and exception policies. We also briefly discuss several methods in achieving several versions of (static and dynamic) separation of duties.

Because our logical formalization of ABAC treats authorization policies about all types of actions in a uniform way, one can consider straightforward extensions of our AC framework to application domains and actions other than access control. Furthermore, one may also utilize the formalization of the high-level structure incorporation with an existing policy language, such as XACML.

References

[1] C. T. Hu, D. F. Ferraiolo, D. R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (abac) definition and considerations," National Institutes of Standards and Technology (NIST), Gaithersburg, Maryland, Special Publication (NIST SP) 800-162, January 2014.

[2] D. Servos and S. L. Osborn, "Current research and open problems in attribute-based access control," *ACM Computing Surveys*, vol. 49, no. 4, pp. 65:1–45, January 2017.

[3] X. Jin, R. Krishnan, and R. Sandhu, "A unified attribute-based access control model covering dac, mac and rbac," in *Data and Applications Security and Privacy XXVI (DBSec 2012)*, ser. Lecture Notes in Computer Science, N. C.-B. et al., Ed. Heidelberg: Springer Verlag, 2012, vol. 7371, pp. 41–55.

[4] X. Jin, "Attribute-based access control models and implementation in cloud infrastructure as a service," Ph.D. dissertation, The University of Texas at San Antonio, San Antonio, TX, May 2014.

[5] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson, "The logic of XACML," in *Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*, Springer. Berlin Heidelberg: Springer, 2011, pp. 205–222. [Online]. Available: https://doi.org/10.1007/978-3-642-35743-5_13

[6] G. Bruns and M. Huth, "Access control via belnap logic: Intuitive, expressive, and analyzable policy composition," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 1, p. 9, 2011.

[7] J. Crampton and C. Morisset, "Ptacl: A language for attribute-based access control in open systems," in *Proceedings of the First International Conference on Principles of Security and Trust POST 2012*, ser. Lecture Notes in Computer Science, P. Degano and J. D. Guttman, Eds., no. 7215. Berlin Heidelberg: Springer-Verlag, 2012, pp. 390–409. [Online]. Available: https://doi.org/10.1007/978-3-642-28641-4_21

[8] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo, "An algebra for fine-grained integration of XACML policies," in *Proceedings of the 14th ACM symposium on Access Control Models and Technologies (SACMAT 2009)*. New York, NY: ACM, 2009, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/1542207.1542218

[9] L. Wang, D. Wijesekera, and S. Jajodia, "A logic-based framework for attribute based access control," in *Formal Methods in Software Engineering (FMSE'04)*, ACM. New York, NY: ACM, October 2004, pp. 45–55, 100045.

[10] X. Zhang, Y. Li, and D. Nalla, "An attribute-based access matrix model," in *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC)*, H. Haddad, L. M. Liebrock, A. Omicini, and R. L. Wainwright, Eds. New York, NY: ACM, 2005, pp. 359–363. [Online]. Available: http://doi.acm.org/10.1145/1066677.1066760

[11] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*. New York, NY: ACM, 2006, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180418

[12] P. Jin and Y. Fang-chun, "Description logic modeling of temporal attribute-based access control," in *2006 First International Conference on Communications and Electronics*, N. Q. Trung, K. Tanaka, and H. Lee, Eds. Los Alamitos, CA: IEEE, Oct 2006, pp. 414–418.

[13] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, "Logical attestation: an authorization architecture for trustworthy computing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 249–264.

[14] J. Jiang, R. Chirkova, J. Doyle, and A. Rosenthal, "Towards greater expressiveness, flexibility, and uniformity in access control," in *In SACMAT '18: Proceedings of the 23th ACM symposium on Acc ess control models and technologies*, 2018.

[15] S. Jajodia, P. Samarati, M. L. Sapino, and V. Subrahmanian, "Flexible support for multiple access control policies," *ACM Transactions on Database Systems (TODS)*, vol. 26, no. 2, pp. 214–260, 2001.

[16] P. A. Bonatti and P. Samarati, "Logics for authorizations and security," in *Logics for Emerging Applications of Databases*. Springer, 2004, pp. 277–323.

[17] S. Godik and T. Moses, "Oasis extensible access control markup language (xacml)," *OASIS Committee Secification cs-xacml-specification-1.0*, 2002.

[18] V. Kolovski and J. Hendler, "Xacml policy analysis using description logics," *Under submission*, 2008.

[19] G.-J. Ahn, H. Hu, J. Lee, and Y. Meng, "Reasoning about xacml policy descriptions in answer set programming (preliminary report)," in *13th International Workshop on Nonmonotonic Reasoning (NMR 2010)*, 2010, pp. 1–10.

[20] J. Crampton, C. Morisset, and N. Zannone, "On missing attributes in access control: Non-deterministic and probabilistic attribute retrieval," in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. ACM, 2015, pp. 99–109.

[21] J. Crampton and C. Morisset, "Monotonicity and completeness in attribute-based access control," in *International Workshop on Security and Trust Management*. Springer, 2014, pp. 33–48.

[22] D. Servos and S. L. Osborn, "HGABAC: towards a formal model of hierarchical attribute-based access control," in *Seventh International Symposium on Foundations and Practice of Security*, ser. Lecture Notes in Computer Science, F. Cuppens, J. García-Alfaro, A. N. Zincir-Heywood, and P. W. L. Fong, Eds., no. 8930. Cham, Switzerland: Springer, 2014, pp. 187–204. [Online]. Available: https://doi.org/10.1007/978-3-319-17040-4_12

[23] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough, and B. Thuraisingham, "Rowlbac: Representing role based access control in owl," in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '08. New York, NY, USA: ACM, 2008, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1377836.1377849

[24] N. K. Sharma and A. Joshi, "Representing attribute based access control policies in OWL," in *Tenth IEEE International Conference on Semantic Computing, ICSC 2016, Laguna Hills, CA, USA, February 4-6, 2016*. Los Alamitos, CA: IEEE Computer Society, 2016, pp. 333–336. [Online]. Available: https://doi.org/10.1109/ICSC.2016.16

[25] M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, *OWL Web Ontology Language Reference*. Cambridge, MA, USA: World Wide Web Consortium (W3C), February 2004. [Online]. Available: http://www.w3.org/TR/owl-ref/

[26] T. Berners-Lee and D. Connolly, "Notation3 (n3): A readable rdf syntax," World Wide Web Consortium (W3C), Tech. Rep., March 2011. [Online]. Available: http://www.w3.org/TeamSubmission/n3/

[27] J. C. John, S. Sural, and A. Gupta, "Attribute-based access control management for multicloud collaboration," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 19, 2017. [Online]. Available: https://doi.org/10.1002/cpe.4199

[28] F. Giunchiglia, R. Zhang, and B. Crispo, "RelBAC: Relation based access control," in *Fourth International Conference on Semantics, Knowledge and Grid, SKG '08, Beijing, China, December 3-5, 2008*. Los Alamitos, CA: IEEE Computer Society, 2008, pp. 3–11. [Online]. Available: https://doi.org/10.1109/SKG.2008.76

[29] M. Evered and S. Bögeholz, "A case study in access control requirements for a health information system," in *ACSW Frontiers 2004, 2004 ACSW Workshops - the Australasian Information Security Workshop (AISW2004), the Australasian Workshop on Data Mining and Web Intelligence (DMWI2004), and the Australasian Workshop on Software Internationalisation (AWSI2004) . Dunedin, New Zealand, January 2004*, Sydney, NSW, Australia, 2004, pp. 53–61. [Online]. Available: http://crpit.com/confpapers/CRPITV32Evered.pdf

[30] D. Lembo, D. Pantaleone, V. Santarelli, and D. F. Savo, "Easy OWL drawing with the graphol visual ontology language," in *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016.*, Menlo Park, CA, 2016, pp. 573–576. [Online]. Available: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12904

[31] H. Liu, C. Lutz, M. Milicic, and F. Wolter, "Updating description logic aboxes." in *KR*, 2006, pp. 46–56.

[32] P. Hitzler, M. Krötzsch, A. Parsia, P. F. Patel-Schneider, and S. Rudolph, *OWL 2 Web Ontology Language Reference Primer (Second Edition)*. Cambridge, MA, USA: World Wide Web Consortium (W3C), December 2012. [Online]. Available: https://www.w3.org/TR/owl2-primer/

[33] R. Verborgh and J. De Roo, "Drawing conclusions from linked data on the web: The eye reasoner," *IEEE Software*, vol. 32, no. 3, pp. 23–27, 2015.

[34] F. Baader and P. Hanschke, "A scheme for integrating concrete domains into concept languages," in *Proceedings of the 12th International Joint Conference on Artificial Intelligence -*

Volume 1, ser. IJCAI'91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 452–457. [Online]. Available: http://dl.acm.org/citation.cfm?id=1631171.1631239

[35] F. Baader, *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[36] R. Brachman and H. Levesque, *Knowledge Representation and Reasoning*. San Francisco: Morgan Kaufmann, 2004.

[37] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Commun. ACM*, vol. 54, no. 12, pp. 92–103, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043174.2043195

[38] J. F. Horty, "Moral dilemmas and nonmonotonic logic," *Journal of Philosophical Logic*, vol. 23, no. 1, pp. 35–65, 1994. [Online]. Available: http://dx.doi.org/10.1007/BF01417957

[39] J. Horty, *Reasons as defaults*. Oxford University Press, 2012.

[40] S. Jha, S. Sural, V. Atluri, and J. Vaidya, "Enforcing separation of duty in attribute based access control systems," in *Information Systems Security*, S. Jajoda and C. Mazumdar, Eds. Cham: Springer International Publishing, 2015, pp. 61–78.

[41] S. Jha, "Specification and verification of separation of duty constraints in attribute-based access control," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 4, pp. 897–911, 2018.