Journal of Information Security and Applications 44 (2019) 49–63

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

Modeling adaptive access control policies using answer set programming

Sara Sartoli[a], Akbar Siami Namin[b,*]

[a] Department of Computer Science and Information Systems, University of North Georgia, Dahlonega, Georgia, USA
[b] Computer Science Department, Texas Tech University, Lubbock, Texas, USA

ARTICLE INFO

Article history:
Available online 27 November 2018

Keywords:
Access control
Inference mechanism
Answer set programming
Policies
Exception handling
Conflict

ABSTRACT

Many of the existing management platforms such as pervasive computing systems implement policies that depend on dynamic operational environment changes. Existing formal approaches for automatically enforcing access control policies are primarily expressed in conventional logic programming, also known as monotonic logics, e.g., First Order Logic (FOL). The major issue with monotonic logics is that they are not devised to invalidate initial believes in the light of further observations. This limitation makes these traditional logical approaches less suitable for modeling and analyzing context-aware access control policies, where exceptional policies are introduced incrementally and adaptively during runtime. The inability to invalidate initial policies when an exception needs to be enforced might result in inconsistencies and violations that need to be resolved manually by human entities. To address the problems with conventional logical approaches and more importantly prevent such inconsistencies, this paper presents a non-monotonic logic-based reasoning scheme for modeling and analyzing adaptive access control policies. In the proposed formalism, unavailable context data and incomplete access control policies can be explicitly expressed. To do so, the paper distinguishes three kinds of policies: default, context-dependent and exception policies. The proposed formalism is based on Answer Set Programming (ASP), a non-monotonic logic programming language that allows elegant representation of unavailability of context data in adaptive systems. We devise non-monotonic policy inference rules such that, when exception policies are defined, they take precedence over default and context-dependent policies automatically. The results of two case studies are reported to demonstrate the feasibility of the proposed policy representation scheme compared to the Organizational-Based Access Control (OrBAC) model.

1. Introduction

Modern policy-based management systems must implement and enforce policies that depend on the contexts and dynamics of underlying operational environments. The primary goal of implementing an adaptive security mechanism is to enable management systems adjust their protection strategies in the presence of changes occurred in their operational environment [1,2]. Hence, a typical adaptive access control framework must offer an effective mechanism in order to deal with *exceptional* situations that often occur due to unexpected events or behaviors occurred in the contexts of dynamic systems and thus enable runtime access control decisions to mitigate the security risks such as information expositions caused by the changes [3–5].

To deal with context changes and thus make informed decision in the presence of exceptional situations, a self-adaptive software system must address a number of daunting challenges such as [6]:

1. reasoning based on “imperfect” context data, i.e. unavailable or noisy context data;
2. reasoning based on “incomplete” set of policies, e.g., lack of specified policies for some subjects, objects or environmental contexts;
3. resolving inconsistencies caused by runtime context changes in dynamic systems; and
4. resolving inconsistencies caused by exception policies that are added incrementally to the knowledge-base and are in apparent conflicts with predefined default and context-dependent policies.

In this paper, we use the phrase “Adaptive Access Control Scheme” to describe the capability of underlying (i.e. backend) reasoning engines of policy-based management systems to actively refine policies in response to changes occurred in an operational

* Corresponding author.
E-mail address: akbar.namin@ttu.edu (A.S. Namin).

https://doi.org/10.1016/j.jisa.2018.10.007
2214-2126/

environment. More specifically, we focus on the ability of management systems to override predefined policies (i.e., default and context-dependent policies) automatically when exception policies (i.e., user-level policies that might be injected incrementally to the knowledge-base) must be enforced instead of predefined policies. This ability is particularly important because the inability of management systems to override predefined policies often introduce inconsistencies, unintended behavior or undetermined access decisions in dynamic management systems. By Default policies, we refer to general policies, which are predefined and thus are applicable to normal executions of the systems. By context-dependent policies, we refer to policies, which are associated with some expected conditions such as time and locations. Furthermore, default and context-dependent policies are defined at the design stage of a system. Whereas, exception policies refer to the policies that are not predefined and are injected incrementally at runtime in unexpected or unknown situations or when an abnormal behavior is detected [7]. Exception policies can also be withdrawn at runtime when the exceptional situation is resolved.

This paper proposes an automated reasoning scheme based on Answer Set Programming (ASP) that explicitly represents imperfect context data and incomplete set of policies. The scheme separates the specification of default, context-dependent and exception policies. The proposed approach benefits from non-monotonicity characteristic of ASP, where defaults and exceptions are explicitly represented using "negation as failure". While enforcing access control policies, the non-monotonicity characteristic enables overriding predefined policies when environmental contexts change or exception policies must be enforced. This feature prevents inconsistencies that often occur due to context changes in dynamic systems and thus require to be resolved manually. The scheme also offers a means to default decision-making, when context data are imperfect or the policy set is incomplete.

The proposal of this research has been presented as a poster [8]. This paper is an extended and enhanced version of our short paper [6] where we introduced the core elements of an adaptive access control scheme. This paper also explores the problem of incomplete access control and conflicting policies. Moreover, the paper presents a detailed research problem and an additional case study in comparison to its shorter version.The main contribution of this paper is to build an automated reasoning scheme that enables reasoning about expressive access control policies. The reasoning scheme can be used as a backend for security management systems. The key contribution can be broken down as follows:

- A formal logic-based representation of context-aware access control, in which imperfect context data and incomplete access control policy sets are explicitly taken into account and expressed.
- A semantic model that distinguishes three types of policies: i) default, ii) context-dependent, and iii) exception policies.
- A formal approach to devising non-monotonic policy inference rules and automate prioritization of exceptions policies over predefined regulations.
- The implementation of the proposed adaptive security reasoning using off-the-shelf efficient ASP solvers and
- The results of two case studies with the purpose of assessing the feasibility and expressiveness of the presented approach compared to context-aware access control models based on First-Order Logic (FOL).

The rest of this paper is organized as follows: Section 2 presents a detailed research problem and motivates the needs for an adaptive access control. Section 3 briefly introduces the notation and semantic of Answer Set Programming (ASP). Section 4 reviews the related work. The ASP-based reasoning scheme is presented through Section 5.1. Concerns regarding incomplete policy sets and conflict management are discussed in Section 7. Section 8 reports two case studies and Section 10 concludes the paper and highlights the future work directions.

2. Motivation

This section motivates the needs for an adaptive access control modeling approach. To do this, we briefly review Organization-Based Access Control (OrBAC), one of the well-known context-aware access control models. This type of access control model is usually expressed using monotonic logic, in particular First Order Logic(FOL). We argue that even though FOL-based reasoning can be useful for design-time policy specification of context-aware access control models, it is not properly devised for modeling adaptive reasoning where exceptions must be handled during runtime. We also highlight deficiencies in expressing adaptive access control through a few of examples.

2.1. Organization-Based Access Control

Organization-Based Access Control (OrBAC) is a context-aware access control model which is an extension of popular Role Based Access Control [9] model represented in first-order logic [10]. One of the key features of OrBAC is that it enables expressing security policies at a high-level of abstraction, i.e., abstract policies. As a result, the concrete policies pertinent to the operation of the underlying system are then inferred from the abstract policies.

Abstract policies are specified based on three abstracted entities: i) roles, ii) activities, and iii) views. In an analogous way, concrete policies are expressed based on three concrete-level entities: i) subjects, ii) actions, and iii) objects. Hence, in order to define policies at an abstract level, subjects, actions and objects are abstracted into roles, activities and views, respectively.

Abstract Policies. Basically, in OrBAC, the abstract access control policies can be expressed through three types of facts that are represented by following predicates:

- *permission(org, role, activity, view, context)*, stating that in an organization *org*, anyone, whose role is classified as *role*, is allowed to perform the activity *activity* on the view *view* when the given context *context* holds.
- *prohibition(org, role, activity, view, context)*, stating that in an organization *org*, anyone, whose role is classified as *role*, is disallowed to perform the activity *activity* on the view *view* when the given context *context* holds.
- *obligation(org, role, activity, view, context)*, stating that in the organization *org*, anyone, whose role is classified as *role*, is obliged to perform the activity *activity* on the view *view* when the given context *context* holds.

Concrete Policies. An OrBAC model represents concrete level policies, i.e., policies in a lower level of abstraction that are inferred from abstract policies, through the following three predicates:

- *isPermitted (org, subj, action, obj)*, stating that in an organization *org*, the subject *subj* is allowed to perform the action *action* on an object *obj*.
- *isProhibited (org, subj, action, obj)*, stating that in an organization *org*, the subject *subj* is disallowed to perform the action *action* on an object *obj*.
- *isObliged (org, subj, action, obj)*, stating that in an organization *org*, the subject *subj* is forced to perform the action *action* on an object *obj*.

MotOrBAC [11] is a user interface that enables specifying and editing OrBAC model policies. The tool is capable of automatically inferring concrete policies given certain abstract policies along

with the association rules between the abstract and concrete entities. It is important to note that the expression of policies in MotOrBAC is based on a first-order logic (FOL) semantic. For instance, MotOrBAC uses the following FOL-based rule to infer the concrete permission policies from the abstract permission policies:

$$isPermitted\ (Org, Subj, Action, Obj) :- \quad (1)$$
$$permission(Org, Role, Activity, View, Context), \quad (2)$$
$$empower(Org, Subj, Role), \quad (3)$$
$$consider(Org, Action, Activity), \quad (4)$$
$$use(Org, Obj, View), \quad (5)$$
$$hold(Org, Subj, Action, Obj, Context). \quad (6)$$

This inference rule states that in an organization *Org*, the subject *Subj* is permitted to perform the action *Action* on an object *Obj* (line 1) if:

1. a permission is specified for any user whose role is *Role* to do the actions abstracted into the activity *Activity* on the objects abstracted into the view *View* in the context *Context* (line 2),
2. the subject *Subj* is empowered as the role *Role* (line 3),
3. the Action *Action* implements (considers) the activity *Activity* (line 4),
4. the object *Obj* is an instance of a view *View* (line 5), and
5. the context associated with the concrete entities is currently held in the organization *Org* (line 6).

The FOL-based rules to infer *isObliged* and *isProhibited* can be expressed in analogous ways.

2.2. The adaptability issues of Organization-Based Access Control

This section presents two examples through which the deficiency of OrBAC-based reasoning scheme, as an instance of monotonic logic-based reasoning schemes, is illustrated. The examples justify the need for an adaptive reasoning scheme for enforcing context-aware access control policies. The prospective scheme is required to be more sensitive to context changes and exception policies and hence thus naturally adapt to frequently changing environments.

Example 1 Handling Exception Policies in OrBAC. Consider a city hospital where OrBAC-based policies are devised to enforce access controls. A permission policy can be expressed in terms of OrBAC-based model as follows [10]:

- $permission(city\_hospital, intern, handle, medical\_file, morning)$.

This policy states that at City Hospital, any intern is allowed to handle medical files in the morning.

Now, let us assume a scenario where i) Bob is appointed as an intern, ii) the action **read** implements the *handle* activity, and iii) Patrice-medical-file is an instance of *medical_file*. Bob is an exceptional intern, whose access to Patrice' s data is restricted due to personal conflicts. Therefore, Bob *exceptionally* must be disallowed to read Patrice' s medical file, even though, according to the predefined abstract policy, he should be able to have access to any medical files including Patrice medical data.

Given the fact that it is unrealistic and infeasible to predefine all possible policies for every possible context, the policy sets are usually incomplete. To ease handling such cases, an access control model must enable integrating exceptional cases into the knowledge-base incrementally at runtime. Because of the restrictions inherited by monotonic logics, OrBAC-based representations of policies cannot enable integrating exceptional concrete policies, such as the Bob and Patrice' s conflict case. The reason is that, even if exception policies are added to the policy set at runtime, the consequences inferred by the abstract policies cannot be overridden without manual administration, which is impractical in dynamic environments.

Let us assume that OrBAC has been extended in such a way that it allows expressing concrete policies along with the abstract ones. Thus, the following exception policy could be integrated into the existing policy set:

- $isProhibited(city\_hospital, bob, read, patrice\_medical\_file)$.

Integrating this exception policy to the pool of policies causes a conflicting problem between the concrete permission policy that is inferred from the abstract permission policy and the concrete exception prohibition policy. According to the abstract permission policy Bob, who is an intern at the hospital, is allowed to read medical files and thus is still permitted to read Patrice's medical file. On the contrary, according to the newly added exception prohibition policy, Bob is prohibited to read Patrice's medical file. In fact, the OrBAC-based knowledge-base does not adapt to the newly added exception and still infers $isPermitted(city\_hospital, bob, read, patrice\_medical\_file)$, which is in apparent conflict with the added exception policy.

As a matter of fact, extending the OrBAC-based policies to represent exceptional cases does not imply suspending or bypassing the abstract policies' consequences. This issue causes an inconsistency between the exception and inferred concrete policies. A possible solution to resolve this type of conflict is through i) assigning *priorities* to policies, or ii) defining clear *separation between abstract entities* [12]. Without a rigorous representation of policies, this issue adds additional complexities to the policy management systems that operate in highly dynamic systems with exceptions.

Example 2 Difficulties in Reasoning When Context Changes. This example illustrates the deficiency of FOL-based reasoning schemes in dynamic environments where OrBAC-based predicates and inference rules are used to express adaptive scenarios. Consider the following two policies in a research lab at City University (CU) where policies are expressed through an OrBAC-based model.

- $prohibition(cu, undergrad\_student, access, research\_lab, default\_ctx)$.
- $permission(cu, undergrad\_student, access, research\_lab, accompanied\_by\_grad\_student)$.

The first policy states that: normally (i.e., in default context), any undergrad student is disallowed to access a research lab at CU; Whereas, the second policy states that at CU any undergrad student is allowed to have access to the research lab if a graduate student is already present in the lab.

Let us consider a scenario where Mary, who is an undergraduate student, attempts to enter EC202 when no graduate student is present in the lab. In this scenario, the action *enter* is abstracted into an access activity, and the EC202 lab is an instance of *research_lab*. With respect to the specified abstract policies and the association rules between the concrete and abstract policies, the OrBAC FOL-based inference rule infers the following concrete policy:

- $isProhibited(cu, mary, enter, ec202)$.

Now, assume another scenario where Mary again attempts to enter EC202 at a later time. Alice, a graduate student, is already in the lab and therefore the context *accompanied_by_grad_student* holds. Since a graduate student now accompanies Mary in the lab, OrBAC FOL-based inference rule infers the following concrete policy:

- *isPermitted*(*cu, mary, enter, ec*202).

It is important to note that, because FOL is used in representing the policies, even after inferring *isPermitted(cu, mary, enter, ec202)*, the concrete policy *isProhibited(cu, mary, enter, ec202)* is not invalidated and thus there is an apparent conflict between these two concrete policies. As a result, the security administration needs to manually resolve this type of conflict at the design time.

A key feature missing in FOL-based reasoning schemes that makes them less attractive in modeling dynamic domains is the lack of reasoning capability in invalidating earlier consequences and believes when changes occur in the contexts or an additional piece of knowledge is gained and added to the knowledge-base. This problem is caused by the monotonicity characteristic of first-order logic in which new observations are never invalidated earlier consequences and therefore it is always assumed that the underlying system is complete and certain.

This paper proposes a formal scheme that employs non-monotonic logic for implementing context-aware access control policies. More specifically, as a non-monotonic declarative language employed in our scheme, Answer Set Programming (ASP) offers an elegant way to represent the knowledge that might be even incomplete or unavailable ahead of time yet make informed decision through uncertain reasoning inherited from its non-monotonic characteristic. The non-monotonicity of ASP makes this declarative logic programming language a suitable tool for automated reasoning for enforcing security policies and requirements in dynamic environments during runtime.

3. Answer Set Programming (ASP)

Answer Set Programming (ASP) [13,14] is a non-monotonic, declarative, and logic programming language. ASP has its origin in default logics [15] and constraint satisfaction paradigm. It is based on stable models [16], also known as answer set semantics of logic programming. What distinguishes ASP from other non-monotonic logic-based approaches is that it is supported with a good number of well-developed and efficient solvers and grounders such as Clasp [17], DLV [18], and Gringo [19]. ASP has been successfully used for reasoning efficiently with incomplete knowledge and also for representing and reasoning about dynamic domains [20,21]. It has recently been used to resolve challenges in different application areas such as robotics, computational biology and e-medicine [22]. In the context of access control, ASP has been used by researchers for XACML policy verification purpose [23–25]. Furtermore, it has been used to provide a simple authorization and obligation language as well as ASP-based compliance checking algorithms [26].

The basic building blocks of ASP programs are atoms. Atoms are factual statements that are evaluated to be true or false. Literals are either atoms or the negation of atoms. An atom $p(t_1,\ldots,t_n)$ is a ground atom if all of t_is are constants (i.e. none of them is a variable). Ground literals are either ground atoms or their negations. ASP rules are ordered pairs in the general form of:

$$Head :- Body$$

The *Head* and the *Body* are finite sets of literals. Using :– in rules means that the ASP rules are executable. In this paper, we use a subset of ASP programs, in which every rule after expanding *Head* and *Body* is in the form of:

$$a :- b_1,\ldots,b_i, not\ c_1,\ldots,not\ c_j$$

The rule states that *a*, (i.e., the *Head*) is true if all of the atoms in the *Body* (the right side of the rule) are true. In other words, *a* is true if $b_1,\ldots,b_i$ are believed, whereas, $c_1,\ldots,c_j$ are not believed. The symbol *not* is called "*default negation*" or "*negation as failure*." The *not* c_i notation can be interpreted as: *there is no reason to believe that* c_i *is true*. Unlike the classical negation, *not* does not imply that c_i is believed to be false. Rather it implies that there is not enough evidence to believe in c_i (i.e., the value of c_i is not available). ASP facts are expressed as rules with empty bodies as follows:

$$a :-$$

The empty *Body* is always evaluated to be true. In this paper, we omit :– when expressing facts.

An ASP program, i.e., a set of ASP rules, can semantically be viewed as a specification for answer sets. Answer sets can be described as sets of beliefs that can be associated with an ASP program. Answer sets are represented by a collection of ground literals. Forming answer sets are guided by the following set of informal principles [20]:

1. Believe in the head of a rule if its body is believed.
2. Do not believe in contradictions (i.e., a and -a).
3. Believe nothing if it it not forced to be believed.

In the rest of this section, we illustrate the principles together with the notion of classical and default negation.

Example 3.2.1. [A Program with Classical Negation.] Consider the following ASP program with two rules:

$q :- -p$ "If it is believed that p is false, then q must be believed"
$-p$ "It is believed that p is false"

In this example, the second rule is a fact and facts are always believed, since their body is a collection of empty literals. Therefore, based on first principle, it is believed that p is false. The same principle is applicable to the first rule enforcing us to believe q. Therefore, the answer set $\{-p, q\}$ can be associated with the rules of the program.

Example 3.2.2. [A Program with Default Negation.] Sometimes reasoning can be performed based on the absence of information. For example, Alice, who is a nurse, might be permitted to read John's records, who is a patient, while lacking the evidence to the contrary (i.e. by default). Such reasoning can be captured by default negation and can be represented by an ASP program that contains the following type of rules:

$q :- not\ p$ "If there is no reason to believe p, then q must be believed "

Since there is no rule in the program that has p on its head, nothing forces to believe p. So, by principle 3, the answer set does not contain p. Consequently, to satisfy the only rule of the program, since p is not believed q must be believed. Therefore, $\{q\}$ is the answer set of the program.

ASP has several features that makes it a suitable tool to express evolving knowledge in dynamic systems:

- **Explicit and distinct Representations of Default and Exception policies.** The elegance and expressiveness of ASP in representing defaults and exceptions by the means of negation as failure offers two valuable benefits to modeling access control systems:
  - *Reasoning with Imperfect Data*. This feature enables making informed decisions about access control permissions when context data are imperfect ahead of time or the set of policies is incomplete and evolving. In such cases, privileges can be granted based on defaults (i.e., context defaults are represented through the general abstract policies) [8].
  - *Resolving Conflicts and Dealing with Inconsistencies*. Representing defaults and exceptions and invalidating defaults, when exceptions are added to the knowledge base,

help avoiding conflicts between the default and context-dependent (i.e., predefined policies) on one side and the exception policies on the other side naturally, as illustrated in motivating Examples 1 and 2.

- **A Declarative Way to Program and Knowledge Representation.** Declarative programming is a paradigm, in which a programmer intends to describe a problem (i.e., what) rather than the solution (i.e., how). Using declarative languages, policies and requirements can be added incrementally to the knowledge base regardless of being concerned about their side effects and potential inconsistencies. By enabling one to express what restrictions need to be enforced rather than describing how to enforce them, this feature supports adding exceptional policies incrementally at runtime.

4. Related work

4.1. Access control models

Traditional access control models such as Discretionary Access control (DAC) [28], Mandatory Access Control (MAC) [29], and Task-Based Authorization Controls (TBAC) [30] are restricted to modeling permission controls using static policies. Unlike traditional access control modeling schemes, Role Based Access Control (RBAC) [9] enables expressing policies for dynamic systems. RBAC policies are defined based on roles and privileges (i.e.,permissions). Furthermore, RBAC allows coping with operating environment changes by changing user roles when privileges need to be restricted or relaxed. Bertino et. al., introduced Temporal Role-based Access Control (TRBAC) model [31], which extends RBAC to support periodic time triggering roles. However, similar to other traditional access control models, TRBAC also does not allow specifying contexts.

As indicated by Kuhn et. al., RBAC has been criticized for difficulties in modeling role-engineering process and have inadequate support for operating environment context such as time of day and location [27]. The criticisms gave birth to access control models that allow specifying conditions (i.e., contexts) such as OrBAC and Attribute Based Access Control (ABAC) [32]. OrBAC extends RBAC with the notion of prohibition policies and contexts and is formalized in first order logic. Standard implementation of ABAC is eXtensible Access Control Markup Language (XACML) which defines a policy language and an architectural scheme for expressing ABAC [33]. In addition to the RBAC model elements, XACML language allows specifying prohibition (using Deny effect) and contexts (using Condition construct). XACML has been complemented by some logic-based specification approaches that offer formal verification services for XACML policies. Context-aware access control sometimes operates under the assumption that it is possible to predefine access control policies in all possible contexts. However, some of the contexts such as emergency can never be fully defined. The access control in unanticipated conditions are usually denied by default [4].

Building upon the reviewed access control models and having into consideration the reviewed criticisms, this paper proposes to model access control policies within three classes: 1) default policies (similar to RBAC policies but extended with prohibition), 2) context-aware policies (similar to OrBAC policies), and 3) exceptions policies (user level policies to model unanticipated contexts). Once policies are modeled, they can be validated and transformed to policy languages such as XACML. In the proposed model, adding prohibition policies and also the notion of context to RBAC elements enable utilizing denial and condition constructs of XACML features. It also adds exception policies as well as capability to modeling incomplete context data to OrBAC (See Section 6).

4.2. Logic-based approaches to reason about XACML policies

XACML is the standard implementation for ABAC that incorporates a rich set of features and constructs and allows expressing policies and specifying policy combination mechanisms. However, according to several research, specifying policies in XACML is a difficult and error-prone task [34–36]. In other words, specifying correct and efficient policies in XACML depends on policy makers decisions [35,37]. For instance, as indicated by [37] “existing policy structuring mechanisms do not prevent efficiency problems that are caused by bad specifications of policies, especially when policies evolve over time.” Furthermore, XACML does not have a formal semantic. However, since most of the XACML constructs have a declarative flavor, researchers formalize the required constructs and map them to logical semantics which also takes into account the policy verification and validation.

To assist policy developers in order to detect and resolve human errors in XACML policies, formal logic-based verification and analysis tools and techniques have been developed [34,36,38,40]. Fisler et. al., [36] present a software suite for analyzing RBAC access control policies, expressed in XACML. In the software suite, each XACML policy is represented as a multi-terminal binary decision diagram (MTBDD) model in first order logic. The policies are then combined by MTBDD-based combining algorithms that implement the XACML combining algorithms. Safety properties are represented in the Scheme programming language. The suite has two main components: 1) a verification component that takes a policy set and a formal property as inputs and determines whether the policy satisfies the property; and 2) a change-impact analysis component that takes as inputs two policies and spans a set of changes in order to identify effects of policy changes.

Hughes and Bultan [34] use a SAT solver to check the satisfaction of some boolean logic based properties. Turkmen et. al., [38] develop a SMT-based policy analysis framework that automatically translate XACML policies into the corresponding formal specification of policies [39]. The specification enables analysis against a wider range of properties including non-boolean properties that are usually left uninterpreted using the SAT solver based approaches. The SMT-based approach also improves analysis performance in comparison to SAT solvers.

Crampton et. al., [40] address the problem of missing attribute values in analyzing attribute-based access control policies initially expressed in PTaCL and XACML. The solution proposed by Crampton et. al., is based on non-deterministic evaluation. In particular, they use PRISM, a probabilistic model checker, to simulate the non-determinism of retrieving attributes.

The proposed reasoning scheme, presented in this paper, benefits from expressiveness of ASP to model and analyze access control policies. In particular, using default negation we can define elegant context definitions (e.g., modeling topology of operating environment as a complex context [41]) and constraint about non-existence of context data and policies (e.g., scenarios in [8]). One of the distinguishable aspects of our scheme compared to some other non-monotonic approaches is the utilization of efficient and available ASP solvers.

5. Adaptive reasoning scheme for context-aware access controls

This section presents a semantic model that enables making informed decision with the goal of enforcing adaptive context-aware access control policies.

5.1. Policy model

We present the policy model using the formalization offered by the traditional Role-Based Access Control (RBAC) [9] and its

context-aware variation, OrBAC [10]. The basic key elements are described in terms of the following sets:

- **Roles (R)**, where set R is a finite set of m roles $R = \{r_1, \ldots, r_m\}$. A role is defined according to a user' job description, e.g., intern.
- **Users(U)**, where set U is a finite set of n users $U = \{u_1, \ldots, u_n\}$. A user is an entity that requests having access to valuable assets, e.g., Mary.
- **Assets (AS)**, where set AS is a finite set of p valuable objects $AS = \{as_1, \ldots, as_p\}$. An asset is an instance of a protected valuable object, e.g., a confidential data container.
- **Actions (AC)**, where set $AC = \{ac_1, \ldots, ac_q\}$ is a set of q operations defined on assets. Actions are basic operations allowable on assets, e.g., reading confidential data.
- **Context Expressions (CE)**, where the set CE is a finite set of o context expressions $CE = \{ce_1, \ldots, ce_o\}$. Contexts are concrete environmental conditions that have effects on access control decisions, e.g., morning time.
- **Exception Identification Numbers (ID)**, where the set ID is a finite set of o identification numbers $ID = \{ID_1, \ldots, ID_o\}$. Identification numbers are numbers that are associated with exception policies and enable distinguishing different occurrence of exceptions from each other.

Using the above sets, we are now in a position to describe the syntax and informal semantic of relations among these sets that will be used in the proposed reasoning scheme. The user role assignment is expressed as follows:

- **User Assignment (UA)**: $UA \subseteq U \times R$ is a many to many relationship imposed on the sets of users U and roles R. This relationship is used to associate users with their roles in the system. An ASP fact such as $ua(user_1, role_1)$ states that user $user_1$ is associated with role $role_1$ and is expected to perform the duties described for $role_1$.

On the other hand, the default policies can be expressed as follows:

- **Default Permission (dPrm)**: $dPrm \subseteq R \times AC \times AS$ is a ternary relationship imposed on the sets of roles R, actions AC and assets AS. An ASP fact such as $dPrm(role, action, asset)$ states that normally and by default, any user with the assigned role $role$ is allowed to perform action $action$ on asset $asset$.
- **Default Prohibition (dPrh)**: $dPrh \subseteq R \times AC \times AS$ is a ternary relationship imposed on the sets of roles R, actions AC and assets AS. Similarly, an ASP fact such as $dPrh(role_1, action_1, asset_1)$ states that normally and by default, any user with the assigned role $role_1$ is disallowed to perform the action $action_1$ on the asset $asset_1$.

In an analogous way, the context-dependent policies are expressed as follows:

- **Context-Dependent Permission (cdPrm)**: $cdPrm \subseteq R \times AC \times AS \times CE$ is a relationship imposed on the sets of roles R, actions AC, assets AS, and context expressions CE. An ASP fact such as $cdPrm(role_1, action_1, asset_1, ce_1)$ states that any user, whose role is $role_1$, is allowed to perform the action $action_1$ on the asset $asset_1$ when the context expression ce_1 holds, i.e., is evaluated to be true.
- **Context-Dependent Prohibition (cdPrh)**: $cdPrh \subseteq R \times AC \times AS \times CE$ is a relationship imposed on the sets of roles R, actions AC, assets AS, and context expressions CE. An ASP fact such as $cdPrh(role_1, action_1, asset_1, ce_1)$ states that any user, whose role is $role_1$, is disallowed to perform the action $action_1$ on the asset $asset_1$ when the context expression ce_1 does not hold, i.e., is valuated to be true.

Similarly, the exception policies are expressed as follows:

- **Exception Permission (exPrm)**: $exPrm \subseteq U \times AC \times AS \times ID$ is a relationship imposed on the sets of Users U, actions AC, and assets AS. An ASP fact such as $exPrm(user_1, action_1, asset_1, exceptionId)$ states that the user $user_1$ is exceptionally allowed to perform the action $action_1$ on the asset $asset_1$. The $exPrm$ is associated with an $exceptionId$ that allows distinguishing different occurrences of exceptions from each other.
- **Exception Prohibition (exPrh)**: $exPrh \subseteq U \times AC \times AS \times ID$ is a relationship imposed on the sets of Users U, actions AC, assets AS, and Identifiers ID. An ASP fact such as $exPrh(user_1, action_1, asset_1, exceptionId)$ states that the user $user_1$ is exceptionally disallowed to perform the action $action_1$ on the asset $asset_1$. The $exPrh$ is associated with an $exceptionId$ that allows distinguishing different occurrences of exceptions.

Note that in this paper, the term "exception" is used in two different meanings. We use the "exception policy " in the context of access control policies; whereas, the "ASP-based exceptions" are used in the context and semantic of ASP-based default reasoning. More specifically, by "exception policy", we mean the user-level policies are specified for individuals and added incrementally to the knowledge-base; whereas, by an "ASP-based exception", we refer to an element of the default reasoning that represents a fact or a logical inference that is followed by default negation in an ASP rules. Finally, contexts are defined through logical rules that have *hold* predicates on their heads, i.e., the left part of the rules:

- $holds \subseteq U \times AC \times AS \times CE$ is a relationship imposed on the sets of users U, actions AC, assets AS, and CE. An ASP fact such as $holds(user_1, action_1, asset_1, context_1)$ states that the user $user_1$ is allowed to perform the action $action_1$ on the asset $asset_1$ while the condition $context_1$ holds.

In the following sections, we present an in-depth description of the key concepts such as contexts, default policies, context-dependent and exception policies as well as Incompleteness and conflict management as framed in the presented adaptive reasoning scheme. We employ the convention of using lowercase letters to represent objects and uppercase letters to express variables in order to be consistent with the ASP' s syntax.

5.2. Representing different types of contexts

There is a good body of knowledge on the use of contexts in different context-aware applications (for a good survey see [42] and [43]). These context-aware applications use contexts to perform reasoning and adapt their behaviors in order to response to changes occurred in their environments in which changes are usually detected by sensors. The context data detected by sensors is usually unavailable beforehand. Therefore, it is important for a context-modeling scheme to deal with incomplete and noisy data and support self-esteem reasoning to automate decision-making processes.

An attempt to precisely define a context depends on the application domain. From the perspective of access control models, Kalam et al., [10] defined contexts as specifications of concrete circumstances used to express dynamic access control policies. From the perspective of requirements engineering, contexts can be defined as a partial view of a state of an environment that influence the decisions of the system [44]. For example, we are not interested in the part of states of the world that are uniform because they do not influence the decisions of an access control system. In this paper, we adopt the requirements engineering perspective of the contexts associated with access control policies. To do so, we use the following types of contexts to express access control policies:

- **Basic Entities Contexts**. Conditions that are used to describe a certain attribute of model entities. For example, *laboratoryRecord* can be a context expression, used to describe the type of an asset, i.e., the value of type attribute of an asset is *laboratoryRecord*.
- **Relational Contexts**. Conditions that are used to describe a relationship or interaction between instances of model entities. For example, *attendingPhysician* is a context expression that describes the relationship between the owner of a certain asset and a physician, i.e., an attending physician has been assigned as the physician of the owner of an asset.
- **Environmental Contexts**. Conditions that are used to describe the operating environment. The environmental contexts are independent from basic entities. For example, at a typical hospital, *workingHours* is a context expression that can be used to describe working hours and shifts of the hospital.

In the presented ASP-based reasoning model, atomic contexts are inferred from the ASP facts, where the head of rules are inferred using the *holds* predicates (defined in Section 5.1); whereas, the bodies of rules are facts that describe attributes of the entities or the environment. Compound contexts can be inferred based on atomic contexts, where both the head and the body of the rules include the "holds" predicates. CDR_1 and CDR_2 illustrate the definition of an atomic context and a compound context, respectively.

$$CDR_1.\ holds(User, Action, Asset, workingHours) :- \\ afterTime(8), \\ beforeTime(18), \\ -onDay(saturday).$$

$$CDR_2.\ holds(User, Action, Asset, internPrescriptionHour) :- \\ ua(User, intern), \\ holds(User, Action, Asset, morning), \\ holds(User, Action, Asset, evenDays).$$

CDR_1 states that "workingHour" context holds (for all of triples of $< user, action, asset >$) from 8 to 18 everyday except Saturdays. CDR_2 states that "internPrescriptionHour" context holds for users who are assigned as interns, with the condition that if "morning" and "evenDay" contexts are hold. Note that in this paper, we present only a semantic model to reason about policies that depend on contexts and leave out presenting a context model.

5.3. Default, context-Dependent, and exception policies

Due to the complexity of regulations and security goals in a typical dynamic domain, the number of access control policies can grow substantially. Specifying policies and regulations in large organizations is a very tedious and error-prone task. As a result, inconsistencies in policies may occur due to missing policies for certain contexts or possible conflicts among policies pertinent to various contexts. Furthermore, redundant policies [12] may be introduced due to human errors while intending to prioritizing policies manually. For example, redundant policies might be introduced because of assigning lower-ranked priorities to more specific policies (e.g., policies defined for a short period of time).

To address these problems, we propose to separate access control policies into three classes: 1) default policies, 2) context dependent policies, and 3) exception policies. Default access control policies are expressed in a fashion similar to RBAC policies, which are typically generic and context-independent. Unlike, traditional context-aware access control models that have an implicit prohibition default, we enable policy makers to express defaults explicitly and directly. Context-dependent access control policies are defined for certain contexts. Finally, exception policies are access control policies defined for individual users (syntax and semantic of these classes are presented in Section 5.1). We present an example to illustrate these classes of access control policies.

Example 5.3.1. Consider an academic research group affiliated with a research lab. The lab members consist of advisers and graduate students. There are also visitors who might need to have access to the lab occasionally. The following descriptive default and context-dependent access control policies are specified for individuals, labeled as visitors, at design time:

- **A Default Policy**. A visitor normally is not allowed to enter the "CHE-202" lab (Policy $D1$).
- **A Context-Dependent Policy**. A visitor, however, is allowed to enter "CHE-202" lab during the group meeting time (Policy $C1$).

The following exception policy needs to be integrated and enforced at runtime:

- **An Exception Policy**. John, who is a visitor, is exceptionally allowed to enter "CHE-202" lab (Policy $E1$).

The default, context-dependent, and exception access control policies can be formally expressed through the following ASP-based facts:

- **The Default Policy**. *dPrh(visitor, enter, che-202)*, where the *visitor* is a role, the *enter* is an action, and the *CHE-202* is an asset, i.e., the ASP-based expression of Policy $D1$.
- **The Context-Dependent Policy**. *cdPrm(visitor, enter, che-202, meetingTime)*, where the *visitor* is a role, the *enter* is an action, the *CHE-202* is an asset, and the *meetingTime* is a temporal context expression, i.e., the ASP-based expression of Policy $C1$.
- **The Exception Policy**. *exPrm(john, enter, che-202,1)*, where *John* is a user, the *enter* is an action, the *CHE-202* is an asset, and 1 is an id for the exception policy, i.e., the ASP-based expression of Policy $E1$.

5.4. Inferring concrete policies

Using the ASP's default representation feature, concrete policies can be inferred using the following inference rules:

$$IR_1.\ is\ Permitted(User, Action, Asset) :- \\ dPrm(Role, Action, Asset), \\ ua(User, Role), \\ not\ isProhibited(User, Action, Asset).$$

$$IR_2.\ is\ Prohibited(User, Action, Asset) :- \\ cdPrh(Role, Action, Asset, Context), \\ ua(User, Role), \\ holds(User, Action, Asset, Context), \\ notexPrm(User, Action, Asset, ID).$$

$$IR'_2.\ is\ Prohibited(User, Action, Asset) :- \\ cdPrh(Role, Action, Asset, Context), \\ ua(User, Role), \\ holds(User, Action, Asset, Context), \\ exPrm(User, Action, Asset, ID), \\ withdraw(ID).$$

$$IR_3.\ isPermitted(User, Action, Asset) :- \\ exPrm(User, Action, Asset, ID), \\ not\ withdraw(ID).$$

- Rule IR_1 states that: if i) a default permission policy *dPrm* is specified for any user, whose role is *Role*, to do an action *Action* on an asset *Asset*, and ii) the user *User* is assigned to the role *Role*, and iii) there is no reason to believe that (i.e., the *not*) user *User* is prohibited to perform the action *Action* on the asset *Asset*, then the user *User* is permitted to perform the action *Action* on the asset *Asset*.
- Rule IR_2 states that: if i) a context-dependent prohibition policy *cdPrh* is specified for any user, whose role is *Role*, to do an action *Action* on an asset *Asset* under a specific context *Context*, ii) a user *User* is assigned to the role *Role*, iii) the associated context *Context* holds, and iv) there is no reason to believe that (i.e., the *not*) the user *User* is exceptionally permitted to perform the action *Action* on the asset *Asset* then the user *User* is prohibited to perform the action *Action* on the asset *Asset*. Policies like *exPrm* are user-level policies and might be added incrementally at runtime due to exceptional situations and thus they may invalidate the consequences of predefined policies.
- Rule IR'_2 states that: if i) a context-dependent prohibition policy *cdPrh* is specified for any user, whose role is *Role*, to do an action *Action* on an asset *Asset* under a specific context *Context*, ii) a user *User* is assigned to the role *Role*, iii) the associated context *Context* holds, iv) the user *User* is exceptionally permitted (by exception policy *ID*) to perform the action *Action* on the asset *Asset*, and v)the exception policy with id *ID* is withdrawn, then the user *User* is prohibited to perform the action *Action* on the asset *Asset*.
- Rule IR_3 states that: if i) an exception permission policy with id *ID* is specified for a user *User* to perform an action *Action* on an asset *Asset*, and ii) the exception permission policy with id *ID* is not withdrawn, then a concrete permission is inferred for the user *User* to perform the action *Action* on the asset *Asset*.

It is important to note that using the IR_{1-3} (i.e., using default reasoning for concrete policy inference) exception policies take precedence over context-dependent policies and in turn context-dependent policies take precedence over default policies. This precedence mechanism offers a clean strategy for defining and implementing priorities among policies and thus an intuitive solution to the conflict management and resolution problem, which will be discussed in following sections.

In an analogous way, the concrete prohibition policies are derived through the following ASP-based inference rules:

$$IR_4.\ isProhibited(User, Action, Asset) :- \\ dPrh(Role, Action, Asset) \\ ua(User, Role) \\ not\ isPermitted(User, Action, Asset).$$

$$IR_5.\ isPermitted(User, Action, Asset) :- \\ cdPrm(Role, Action, Asset, Context) \\ ua(User, Role) \\ holds(User, Action, Asset, Context) \\ not\ exPrh(User, Action, Asset, ID).$$

$$IR'_5.\ isPermitted(User, Action, Asset) :- \\ cdPrm(Role, Action, Asset, Context), \\ ua(User, Role), \\ holds(User, Action, Asset, Context), \\ exPrh(User, Action, Asset, ID), \\ withdraw(ID).$$

$$IR_6.\ isProhibitted(User, Action, Asset) :- \\ exPrh(User, Action, Asset, ID), \\ not\ withdraw(ID).$$

- Rule IR_4 states that: if i) a default prohibition policy *dPrh* is specified for any user, whose role is *Role*, to do an action *Action* on an asset *Asset*, and ii) the user *User* is assigned to the role *Role*, and iii) there is no reason to believe that (i.e., the *not*) the user *User* is permitted to perform the action *Action* on the asset *Asset*, then the user *User* is prohibited to perform the action *Action* on the asset *Asset*.
- Rule IR_5 states that: if i) a context-dependent permission policy *cdPrm* is specified for any user, whose role is *Role*, to do an action *Action* on an asset *Asset* under a specific context *Context*, ii) a user *User* is assigned to the role *Role*, iii) the associated context *Context* holds, and iv) there is no reason to believe that (i.e., the *not*) the user *User* is exceptionally prohibited to perform action *Action* on the asset *Asset* then the user *User* is not prohibited to perform the action *Action* on the asset *Asset*. Policies like *exPrh* are user-level policies and might be added incrementally at runtime due to exceptional situations and invalidate the consequences of predefined policies.
- Rule IR'_5 states that: if i) a context-dependent permission policy *cdPrm* is specified for any user, whose role is *Role*, to do an action *Action* on an asset *Asset* under a specific context *Context*, ii) a user *User* is assigned to the role *Role*, iii) the associated context *Context* holds, and iv) there is *exPrh* policy with id *ID*, stating that the user *User* is exceptionally prohibited to perform action *Action* on the asset *Asset* then the user *User* is not prohibited to perform the action *Action* on the asset *Asset*, and v) the exception policy with id *ID* is withdrawn.
- Rule IR_6 states that: if i) an exception prohibition policy with id *ID* is specified for a user *User* to perform an action *Action* on an asset *Asset* and ii) the exception prohibition policy with id *ID* is not withdrawn, then a concrete prohibition is inferred for the user *User* to perform the action *Action* on the asset *Asset*.

6. Management of incomplete policies

In context-sensitive access control models incompleteness may refer to existence of some possible situations for which no explicit policy is devised. In this section, we illustrate an incomplete access control policy set to illustrate how the presented scheme is capable of effectively managing these types of situations.

Example 1 (Incomplete Access Control Policies). Consider the specification of two context-dependent policies devised for an automated health-care system:

- A nurse is allowed to read and write any patient's records from within the hospital's emergency room.
- A nurse is disallowed to read or write any patient's records from outside of the hospital.

At a typical hospital, where this policy set is implemented, an access control decision might not be determined when a nurse requests to read/write a patient's records from the operation room. This happens because none of the policies in the policy set is associated with the operating room area. This example demonstrates an incomplete policy case where policies are not predefined for all possible locations at the hospital.

In traditional access control models, the undetermined access control decisions are usually interpreted as *implicit denials*. However, this may not reflect the desired intention of a policy maker [45]. The absence of explicit default policies in case of incomplete context-dependent policy sets might cause undetermined access decision and thus prevent achieving functional goals. This problem can be addressed by expressing access control policies using

many-valued logics as described in [4,46,47] and performing default reasoning to enable decision making based on defaults [6], when none of the context-dependent policies in the policy set is applicable.

Similarly, in the proposed adaptive access control scheme, we categorize access control policies into three classes: default, context dependent and exception policies. The access control policies are expressed using three different types of predicates (e.g., *dPrm, cdPrm*, and *exPrm*). We use defeasible inference rules (i.e., IR_1 to IR_6 rules in Section 5.4) to enforce default access control policies (e.g, *dPrm*) in the absence of applicable context-dependent policies (e.g., *cdPrm*). Based on the introduced predicates and the defeasible inference rules, access control decisions are made based on default policies in the absence of either context-dependent policies or context data.

7. Conflict management

In dynamic systems, conflicts might occur because of continuously changing conditions in the environment, changing regulations or by human mistakes when devising policies. In the context of access control models, a conflict occurs when, for example, a user is both permitted and prohibited to perform a certain action on the same set of assets, simultaneously. In such cases, there is a dilemma to decide whether to grant or revoke the access requests. This section demonstrates the strength of the proposed adaptive reasoning scheme for eliminating or detecting conflicts among access control policies. In the following sections, first we categorize conflicts among access control policies into two groups of interclass and intraclass conflicts. Then we explain how the instances of each group are detected, eliminated or resolved in the proposed approach.

7.1. Types of conflicts

With respect to the proposed access control scheme and representation, conflicting access control policies can be classified into two subcategories:

- Interclass conflicts. Conflicts among prohibition and permission policies of different classes, e.g., conflicts between a default permission (*dPrm*) and a context-dependent prohibition (*cdPrh*).
- Intra-class conflicts. Conflicts among policies within a class, e.g., conflicts between a context-dependent permission (*cdPrm*) and a context-dependent prohibition (*cdPrh*).

7.2. Interclass conflict management

This section first presents a few illustrative examples to demonstrate interclass conflict type. Then, it describes how the instances of interclass conflicts are eliminated by the proposed adaptive access control representation.

Example 2 (Conflicting Default and Exception Policies). Consider the following default policy developed for a lab in Chemistry department:

- Normally undergraduate advisers are disallowed to enter the Chemistry labs.

Suppose the following policy is added to the policy set upon Dr. Green's request, the lab supervisor, while she is on travel:

- Nancy, the undergraduate adviser, is exceptionally allowed to enter Dr. Green's chemistry lab, while Dr. Green is traveling.

There is an apparent conflict between these two policies that can be resolved by assigning a higher priority to the more specific policy (i.e., in this example the second policy is more specific) [10]. Hence, the final access control decision cannot be decided without giving precedence to one of these policies and thus ignoring the other one. We propose to avoid this type of conflict by the means of "default reasoning" instead of assigning *explicit* priorities to policies manually, which introduces administrative overhead and might lead to unintended behaviors when policy sets grow in size.

In the proposed adaptive access control scheme, there are three separate classes of policies: default, context-dependent, and exception policies. Using the proposed ASP-based inference rules (explained in Section 5.4) exception policies take priority over default policies implicitly (Rules IR1-IR6). In other words, each exception policy, when integrated into the policy set, invalidates default regulations. The idea of overriding default policies is similar to most override strategy in [48] that has been introduced in the context of authorization propagation policies. In most override strategy, authorizations of a super-node is propagated to its sub-nodes if not overridden (nodes represent subjects in a hierarchy of subjects). We use the same idea in a different context, that is overriding default policies when more specific policies must be enforced.

Example 3 (Conflicting Default and Context-Dependent Policies). Imagine a digital library that offers online services for accessing published journal papers and databases to its clients. The access privileges are regulated with respect to the organizations (i.e., where users are accessing the digital library). Consider the following default and context-dependent policies:

- Generally, users are not allowed to download and save scientific papers on their local computers.
- Users accessing from a certain network access point, e.g., People's Community College, are allowed to download and save the published journal papers retrieved from the online database.

An apparent conflict occurs when a user requests to download and save a published journal paper on a local computer through the People's Community College's network access point. The final access control permission cannot be decided unless the context-dependent policy takes precedence over the general policy [10]. This type of conflict also can be eliminated by the means of "default reasoning" rather than assigning *explicit* priorities to policies manually. By separating default and context-dependent policy classes and using the ASP-based inference rules (explained in Section 5.4) context-dependent policies took precedence over defaults implicitly (Rules IR1-IR6). In other words, each context-dependent policy, when defined by policy maker, invalidates default regulations naturally.

7.3. Intraclass policies conflict management

As illustrated in aforementioned examples, the proposed ASP-based reasoning scheme naturally eliminates interclass policy conflicts by the means of default reasoning. Moreover, security systems also need to detect and resolve intraclass conflicting policies.

Intraclass conflicting policies, are referred to policies of the same class (i.e., default, context-dependent, or exception policies) in which a policy permits and another one prohibits the same user to perform the same action on the same set of assets. There has been several works on detection and resolution of this type of conflicts [12,45,49–51]. This type of conflicts can be avoided by *prioritizing* policies and specifying *separation of duty* constraints usually performed by a security administrator [12,50].

In this paper, there is no intention to presents a conflict resolution approach for intraclass policies. Rather, We define and encode rules to detect possible conflicts in the proposed adaptive access control. Once policy conflicts are detected then predefined access control policies can be resolved by the security administrator. However, since exception policies are added incrementally during

runtime, the conflicts among exception policies also need to be detected and resolved without human intervention.

In the following subsections, first we define potential policy conflicts in the proposed adaptive access control scheme. Then, we present ASP rules for detecting the potential intraclass conflicts. Finally, we present an ASP rule for resolving conflicts among exception policies (i.e., a specific form of intraclass conflict that needs to be resolved during runtime).

7.4. Formal definition of conflicting policies

This section defines potential conflicts that need to be detected and resolved in the proposed adaptive reasoning scheme. It is important to note that, a typical *interclass* conflict is eliminated by the proposed ASP-based representation implying that all potential conflicts fall into intraclass conflict category.

Conflicting Default Policies. Let the followings be two default access control policies representing the normal permission and prohibition regulations, respectively:

$dPrm(role_1, action_1, asset_1)$

$dPrh(role_2, action_2, asset_2)$

These are conflicting default policies if all of the following three conditions hold:

1. $action_1 = action_2$
2. $asset_1 = asset_2$
3. For some *user*, the knowledge-base contains both $ua(user, role_1)$ and $ua(user, role_2)$

In the case that all of these conditions hold, a concrete conflict occurs because the user will be both permitted and prohibited to perform the same action on the same set of assets.

Conflicting Context-Dependent Policies. Let the followings be two context-dependent access control policies representing context-dependent permission and prohibition regulations, respectively:

$cdPrm(role_1, action_1, asset_1, ce_1)$

$cdPrh(role_2, action_2, asset_2, ce_2)$

These are conflicting context-dependent policies if all of the following four conditions hold:

1. $action_1 = action_2$
2. $asset_1 = asset_2$
3. For some *user*, the knowledge base contains both $ua(user, role_1)$ and $ua(user, role_2)$
4. For some triple (*user, action, asset*), both ce_1 and ce_2 hold

In the case that all of these conditions hold, a concrete conflict occurs because the user will be both permitted and prohibited to perform the same action on the same asset, simultaneously.

Conflicting Exception Policies. Let the followings be two exception access control policies representing enforced permission and prohibition policies, respectively:

$exPrm(user_1, action_1, asset_1, id_1)$

$exPrh(user_2, action_2, asset_2, id_2)$

We say that *exPrm* and *exPrh* are conflicting exception policies if all of the following conditions hold:

1. $action_1 = action_2$
2. $asset_1 = asset_2$
3. $user_1 = user_2$
4. both *exPrm* and *exPrh* are active policies (i.e., are not withdrawn).

In this case, a concrete conflict occurs because a user will be both permitted and prohibited to perform the same action on the same set of assets, simultaneously.

7.5. Conflict detection

This section presents an ASP rule for detecting each of the potential access control policies defined in Section 7.4.

Generally, a policy maker devises default policies at design time. For example, conflicting default policies can be detected statically using the following rule:

$$CD_1.\ dConflict(Role_1, Role_2, Action, Asset) :-$$
$$dPrm(Role_1, Action, Asset),$$
$$dPrh(Role_2, Action, Asset).$$

This rule states that: if i) a default permission policy *dPrm* is specified for the role $Role_1$ to perform an action *Action* on the asset *Asset*, and ii) a default prohibition policy *dPrh* is specified for the role $Role_2$ to perform the same action on the same set of assets, then there is a conflict between the default permission and prohibition policies devised for role $Role_1$ and $Role_2$, to perform action *Action* on asset *Asset*.

It is possible to consider the Separation Of Duty (SOD) between $Role_1$ and $Role_2$ and prevent assigning the same *User* to $Role_1$ and $Role_2$, simultaneously. To handle SOD, we can revise the conflict detection rule to be the following rule:

$$CD_2.\ dConflict(Role_1, Role_2, Action, Asset) :-$$
$$dPrm(Role_1, Action, Asset),$$
$$dPrh(Role_2, Action, Asset),$$
$$not\ sod(Role_1, Role_2).$$

Similar to default policies, context-dependent policies are authored at design time. Therefore, potential conflicting context-dependent policies can be detected statically using conflict detection rules which is analogous to default policies. Exception policies are added incrementally during runtime. Therefore, conflicts among exception policies must be detected and resolved automatically with minimum human intervention. To do that, we use the following detection rule :

$$CD_3.\ exConflict(User, Action, Asset, ID_1, ID_2) :-$$
$$exPrm(User, Action, Asset, ID_1),$$
$$exPrh(User, Action, Asset, ID_2),$$
$$not\ withdraw(ID_1),$$
$$not\ withdraw(ID_2).$$

This rule states that: if i) an exception permission policy *exPrm* is specified for the user *User* to perform an action *Action* on the asset *Asset*, ii) an exception prohibition policy *exPrh* is specified for the same user to perform the same action on the same asset, and iii) *exPrm* and *exPrh* policies are not withdrawn, then a conflict is detected between exception permission and prohibition policies that allow and at the same time disallow the user *User* to perform the action *Action* on the asset *Asset*. Once a conflict between exception policies is detected exception permission policies should be invalidated. To do so, the inference rule 5.4 can be changed to the following rule:

$$isPermitted(User, Action, Asset) :-$$
$$exPrm(User, Action, Asset, ID_1),$$
$$not\ exConflict(User, Action, Asset, ID_1, ID_2).$$

This rule states that: if i) an exception permission policy is specified for a user *User* to perform an action *Action* on an asset *Asset*, and ii) no conflicting exception policy is derived for the triple (*User, Action, Asset*), then a concrete permission is inferred for the user *User* to perform the action *Action* on the asset *Asset*.

Table 1
The City Hospital abstract policies.

Name	Type	Organization	Role	Activity	View	Context
P1	permission	City hospital	extern	analyze	sample	sample analysis
P2	permission	City hospital	intern	prescribe prescription	vpatient	intern presc_hour
P3	permission	City hospital	medical secreter	prescribe appointment	vpatient	default context
P4	permission	City hospital	nurse	analyze	sample	morning
P5	permission	City hospital	doctor	read	medical file	referent doctor
P6	permission	City hospital	doctor	read	medical file	default context
P7	permission	City hospital	extern	read	medical data	emergency context
I1	prohibition	City hospital	extern	prescribe prescription	vpatient	default context
I2	prohibition	City hospital	extern	handle	medical file	default context
I3	prohibition	City hospital	medical secreter	handle	medical data	default context
O1	obligation	City hospital	surgeon	operate	vpatient	anthesic patient

8. Case studies

To demonstrate the expressiveness and feasibility of the proposed ASP-based reasoning scheme for effectively implementing adaptive access control policies, we report the specification of the introduced ASP-based access control on a hospital case study. The case study is excerpted from [10]. The goal of this case study is to evaluate the capability of the proposed ASP-based reasoning scheme in comparison to OrBAC, as an instance of FOL-based access control model. For this case study, we used SPARC system [52], which provides explicit constructs and allows us to specify objects, relations and their sorts. Declaring sorts allows us to avoid thinking about safety condition in ASP rules. SPARC passes the policy specifications to a DLV solver for generating answer sets and answering queries about policies.

The case study consists of three parts: In the first part, we simulate an OrBAC policy system in terms of our ASP-based representation and compare the results inferred by MotOrBAC (see Section 2.1 for detailed descriptions) with the results obtained by running our ASP-based representation using an ASP solver.

The goal of the first part is to show that our representation is at least as expressive and effective as OrBAC's first-order logic representation. In the second part, we inject an exception policy to both the OrBAC policy set and our ASP-based representation and compare the MotOrBAC and ASP solvers' results, accordingly. Finally, in the third part, we provide scalability measures with regards to the proposed policy inference mechanism when the policy model increase in size.

Note that MotOrBAC is a tool designed for expressing and editing policies at design time; To simulate a runtime situation and condition that might occur in dynamic systems, we inject runtime exception policies. Our intend is to compare the capability of the underlying reasoning schemes in handling exceptional situations.

The studied access control policy set regulates accesses to Electronic Health Records (EHR) in a hospital environment. The policy set, accompanied by MotOrBAC, is an XML-based policy file representing the access control polices in an OrBAC model [53].

The domain consists of three hospitals: i) City hospital, ii) South hospital, and iii) North hospital. The policy file includes 11 abstract access control policies (shown in Table 1) including 7 permissions, 3 prohibitions and 1 obligation to control the accesses of users to the electronic health records. 1 presents the abstract access control policies. The subjects are classified into 13 roles such as secretaries, doctors, and nurses. The subjects perform nine different actions such as read, write, and analyze. There are 6 different types of objects (i.e., valuable assets to be protected), such as medical and administration data in 9 environmental contexts. Context expressions might have various definitions in different organizations. For example, in this case study, the *intern_prescription_hour* context at the South hospital is defined to be from 8:00 - 12:00 am. However, at the North hospital branch the same context expression is defined to be from 2:00 – 6:00 pm.

8.1. Case 1: Orbac vs. ASP-based policy: static policies

Many of the policies specified for the hospital example depend on the temporal properties and contexts. In this case study, first, we used MotOrBAC to simulate the hospital abstract policy set (i.e., to infer concrete policies from abstract policies) at three different times of the day as representatives of instances of morning, afternoon, and evening contexts, respectively. A a result, from 11 abstract policies specified in the studied policy set, MotOrBAC inferred 44, 42, and 39 concrete policies in the morning, afternoon, and evening contexts, respectively. We transformed the XML-based policy set into ASP rules, with similar context setups. Given the 11 abstract policies along with the associations between the abstract and concrete entities, the ASP-solver also inferred 44, 42 and 39 concrete policies. The inferred concrete policies were exactly the same concrete policies as produced by MotOrBAC. We used the following ASP-Based concrete permission inference rule:

$$
\begin{aligned}
&isPermitted(Org, Subj, Action, Obj) :- \\
&\quad Prm(Org, Role, Activity, View, Context), \\
&\quad empower(Org, Subj, Role), \\
&\quad consider(Org, Action, Activity), \\
&\quad use(Org, Obj, View), \\
&\quad holds(Org, Subj, Action, Obj, Context).
\end{aligned}
$$

The obligation and prohibition inference rules were expressed in a similar manner. Note that we did not use default reasoning (i.e., negation as failure in the inference rules) in Case Study 1. Therefore, the true benefit of ASP-based reasoning is not demonstrated in this section. Rather, the intend of this study was twofolds: 1) demonstrating the feasibility of the proposed idea, and 2)

illustrating that ASP-based policies are at least as effective as OrBAC policies.

8.2. Case 2: dynamic policies: Orbac vs ASP-based policy

The purpose of this case study is to compare the effectiveness and expressiveness of OrBAC and the ASP-based policies in a dynamic environment when exceptions exist. We do that by injecting an exception policy to the OrBAC and the ASP-based policies manually.

One of the abstract policies expressed in the City hospital policy file is as follows:

- At City hospital, doctors are normally allowed to modify medical files.

In OrBAC, this regulation is represented through the following fact:

- *permission*(*cityHospital*, *doctor*, *write*, *medicalFile*, *default_ctx*).

According to the hospital policy file, *writeDb*(i.e., write on a database) implements the *write* and *patriceMedicalData* is an instance of *medicalFile*. We added a new subject called Sara, and empowered her as a doctor. However, Sara is an exception and not allowed to perform any modification on Patrice medical data:

- Sara is disallowed to perform the *writeDb* action on *patriceMedicalData*.

This exception policy is semantically equivalent to the following OrBAC concrete policy:

- *isProhibited*(*cityHospital, Sara, writeDb, patriceMedicalData*).

First, we tried to add such an exception policy using the MotOrBAC's user interface. Unfortunately, MotOrBAC does not allow regular users to express non-abstract exception policies directly. Therefore, we injected the exception policy into the XML-based policy file manually. But since the injected policy was not context-dependent, MotOrBAC generated a stopping failure while simulating concrete policies. As the last resort, we tried to inject the exception policy by defining a new context called "exception" as follows:

- In the exception context, Sara is permitted to perform a *writeDb* action on the *patriceMedicalData* object.

Note that the context-dependent concrete policy is not formally defined in an OrBAC model. We injected this runtime exception policy into the policy file to evaluate the capability of MotOrBAC when reasoning with exception policies regardless of its formal semantic model. After simulating the updated policy set in MotOrBAC, both of the following concrete policies were inferred by the MotOrBAC simulator:

- Sara is permitted to perform a *writeDb* activity on *patriceMedicalData*.
- Sara is prohibited to perform a *writeDb* activity on *patriceMedicalData*.

Such an inconsistency is common when reasoning takes place based on monotonic logics. This indicates that the OrBAC system was not able to adapt itself when an exception policy needs to be enforced. To solve this inconsistency MotOrBAC suggests solving concrete conflict manually through prioritizing abstract policies or defining constraints on abstract entities, which is a time consuming and error prone manual task.

As an alternative solution as introduced in this paper, the "negation as failure" can be used to invalidate normal policies automatically in exceptional contexts. We used the following ASP-Based rule to infer concrete permission policies:

$$
\begin{aligned}
&isPermitted(Org, Subj, Action, Obj) :- \\
&\quad dPrm(Org, Role, Activity, View), \\
&\quad empower(Org, Subj, Role), \\
&\quad consider(Org, Action, Activity), \\
&\quad use(Org, Obj, View), \\
&\quad not\ isProhibited(Org, Subj, Action, Obj).
\end{aligned}
$$

$$
\begin{aligned}
&isProhibited(Org, Subj, Action, Obj) :- \\
&\quad cdPrm(Org, Role, Activity, View, Context), \\
&\quad empower(Org, Subj, Role), \\
&\quad consider(Org, Action, Activity), \\
&\quad use(Org, Obj, View), \\
&\quad holds(Org, Subj, Action, Obj, Context), \\
&\quad not\ exPrm(Org, Subj, Action, Obj).
\end{aligned}
$$

$$
\begin{aligned}
&isPermitted(Org, Subj, Action, Obj) :- \\
&\quad exPrm(Org, Subj, Action, Obj).
\end{aligned}
$$

The term after "not" represents exceptions in ASP. In other words, in this rule *isProhibited* concrete policy is an exception to default permission policy and implicitly takes precedence over the default policies.

In a similar manner, the MotOrBAC concrete prohibition and obligation inference rules were modified. We added the exception policy to our ASP-based knowledge-based system. Unlike MotOrBAC, the only concrete policy inferred by the ASP solver was:

- At the *cityHospital*, Sara is prohibited performing the *writeDb* activity on *patriceMedicalData*.

The result of ASP-based policy setting indicates that the proposed ASP-based system adapted itself by invalidating default conclusions when an exception occurred.

Eliminating Conflicts in Case 2. During this case study, we observed that out of 27 potential conflicts detected by MotOrBAC that needed to be resolved manually, 16 of which were potential conflicts between the abstract policies associated with default contexts (i.e., default policies in the proposed reasoning scheme) and abstract policies pertinent to the non-default contexts (i.e., context-dependent policies in the proposed reasoning scheme). Fig. 1 shows some of the conflicts detected by MotOrBAC. However, by invalidating the default policies consequences when the policy set contains context-dependent policies, the proposed ASP-based policy inference mechanism decreases the number of potential conflicts from 27 to 11 potential conflicts.

8.3. Scalability of policy inference mechanism

We performed some scalability experiments with increased policy model sizes on a MacBook Pro-laptop equipped with 2.7 GHz Intel Core i5 processor and memory limit of 8 GB RAM. We used SPARC system [52] for specifying policies and inferring concrete policies from ASP-based policy models. As indicated earlier, SPARC system uses DLV [18] to solve ASP programs and generate answer sets.

To demonstrate the scalability of the proposed policy inference mechanism, we randomly generated policy models similar to those in the case study, with up to 10,000 elements. The elements could be basic such as subjects, roles, assets and contexts. The relations imposed on basic elements could be abstract policies (i.e., permission and prohibition policies), user-assignment predicates, and holds predicates. The policy model and its elements are described in Section 5.1.

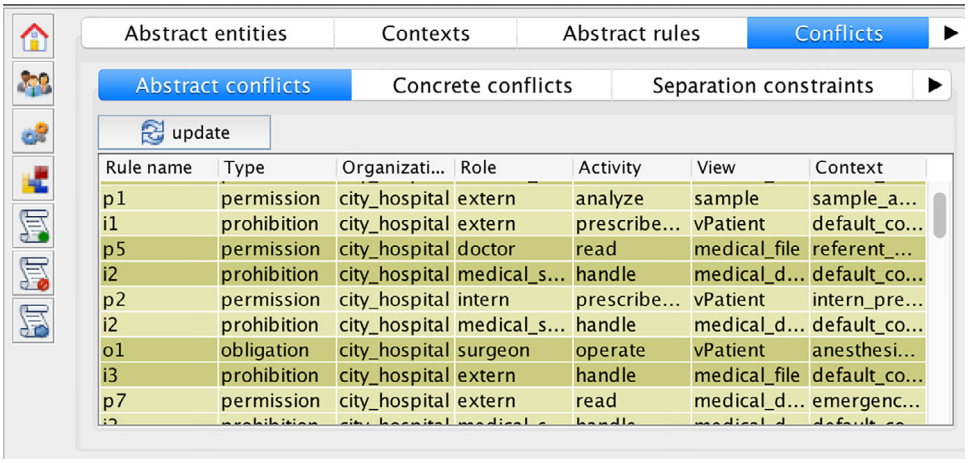

Fig. 1. Potential conflicts derived by MotOrBAC.

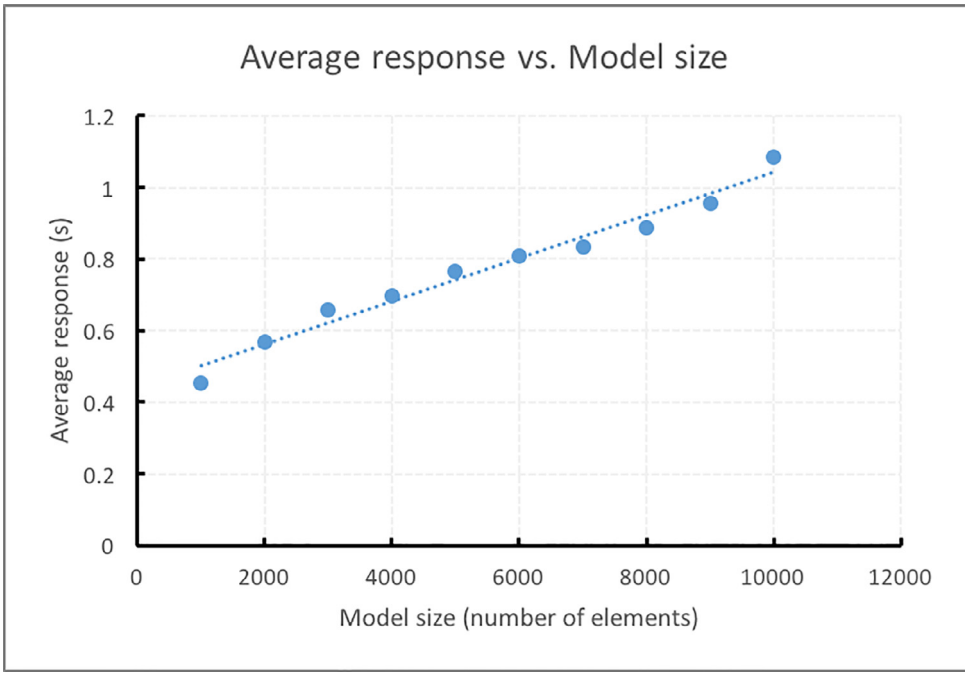

Fig. 2. Scalability of policy inference mechanism.

We increased the policy model size and ran each experiment using SPARC system on the same environment described in the case study. For the purpose of consistency while increasing the model size, each model contained a ratio of 40% subjects, 15% roles, 10% assets, 10% contexts 15% user-assignment predicates and and 10% specified policies. The ratios are based on similar experiment designed and conducted by Bailey et. al., [3]. Additionally, we imposed a *holds* predicate for each context in the generated policy models. To ensure that increasing policy model size (i.e., the input size) fully effect the execution time, the size of inferred policies (i.e., the output size) also increased proportionally to the policy model size. More specifically, in all experiments the number of inferred policies was 15% of input size.

Each experiment was repeated 10 times and captured the average and standard deviation. Fig. 2 shows the average times of the policy inference mechanism in seconds for each policy model size. As the figure indicates, as the model size increases, the average response time increases linearly. Also it is apparent from the figure that policy models that have up to 9000 elements size can be handled easily in less than 1 sec.

9. Discussion

There are two important problems that need to be addressed and the applicability of the proposed formal model needs to be discussed: 1) attribute hiding attacks, 2) context modeling, and 3) the usability of logic-based approaches to security modeling in general.

9.1. Attribute Hiding Attack

Attribute Hiding Attack is a situation, where an attacker deliberately hides attribute values in order to gain a higher level of privileges [47]. Attribute hiding attack is a potential problem for any context-aware access control mechanism, in which decisions rely on monitoring requesters' attributes.The problem of attribute hiding attack has been introduced and addressed in the context of attribute based access controls [38,47].

The introduced access control scheme is also vulnerable to attribute hiding attack. In particular, in the proposed reasoning scheme, if a policy developer chooses to allow access by default

(i.e., specify *dPrm*) and conditionally disallow the access (i.e., specify *cdPrh*), a malicious user can hide the context data, which leads to enforcing default permission policy. The vulnerability to attribute hiding attack can be detected at design time using the following rule:

$$AHD_1.\ attributeHiding(Role_1, Role_2, Action, Asset, Context) :- dPrm(Role_1, Action, Asset), cdPrh(Role_2, Action, Asset, Context).$$

The solution to this problem using the introduced ASP-based approach depends on the decision made by the policy designer. For instance, after the possibility is being detected, the policy developer can decide if she wants to allow access by default or disallow access by default.

9.2. Context modeling

In this paper, context dependent policies are defined on the top of context definition rules. Different types of contexts (e.g., spacial, temporal) have different characteristics. In particular, some of contexts can never hold together (e.g., *morning* and *afternoon*). In some other cases, contexts have hierarchical nature [41,54] (e.g., *in_house* can be considered as a super-node of *in_bedroom*).

Context modeling is an important topic in self adaptive systems but is not the main contribution of this paper. However, the nature of contexts sometimes is an influential and determining factor that affects the occurrence of conflicts. In particular, a $cdPrm(Role_1, Action, Asset, Context_1)$ and $cdPrm(Role_1, Action, Asset, Context_2)$ are potentially conflicting if $Context_1$ and $Context_2$ can be true, simultaneously. A limitation of this work is that, a conflict is detected even if $Context_1$ and $Context_2$ cannot be true simultaneously. Context modeling and addressing this issue will be a part of our future work.

9.3. Usability of logic-based policy security modeling

Logic-based policy schemes have been criticized for being less usable in enabling effective communication with policy makers. To overcome this concern, it would be ideal to allow policy makers to model policies in a higher-level graphical notation and then translate the notations into precise logic-based formalisms. One example of such a high level notation is the graphical representation and user interface offered by STS tool [56]. STS tool allows automated analysis through disjunctive Datalog while providing a security requirements modeling graphical notation (STS-ml) that hides the complexity of the logic and its representation from security experts. Other approaches include providing user interfaces and prompting users to enter access control policies in an interactive graphical environment without delving into developing complex logical rules (e.g., MotOrBAC).

Usability concern is also an issue when specifying XACML policies. Researchers have taken similar approaches to allow simple specification and editing XACML rules independently and then transform the rules to XACML policies. Axiomatic has created such a policy editor that uses Axiomatics Language for Authorization (ALFA) [57]. The Axiomatic policy editor is implemented as an Eclipse plugin. Another popular language is PonderTalk, which allows specifying policies in a higher-level language in Ponder2 environment [55].

Developing similar high level languages, graphical notations, interactive user interfaces and developing optimal policies [58] can be helpful to mitigate the issues related to the usability of the proposed ASP-based formal modeling and thus ease the modeling problem.

10. Conclusion and future work

This paper presented an ASP-based access control scheme to support modeling and implementing adaptive security systems. In order to improve the sensitivity of context-aware access control models to runtime context changes, making policy management easier, and reduce potential conflicts we separated the specifications of default, context-dependent, and exception policies and expressed them as ASP default and exceptions. The proposed model utilized the non-monotonicity and efficiency of ASP solvers to implement defeasible inference rules. We devised non-monotonic policy inference rules to invalidate default policies at runtime when exception or context-dependent policies need to be enforced.

The case study presented in this paper showed that the introduced approach was useful in eliminating conflicts and inconsistencies among default, context-dependent and exception policies that are added incrementally at runtime. The results of case studies demonstrate the applicability of the approach and encourage use of ASP solvers for reasoning about access control policies. We believe that using non-monotonic logics for representing and reasoning about policies is beneficial for self adaptive systems that need to adjust their protection decisions in changing environments.

The proposed reasoning scheme can be extended by including access control model element hierarchies and context models. We are currently, extending this work in three directions. First, we are using structural models and runtime goal models for invalidating predefined policies. The current reasoning scheme can also be extended by variant runtime contextual models. Runtime contextual models need to be transformed and verified at runtime. Therefore transformation and verification of each of new runtime models introduces a new possible research direction.

Acknowledgement

This research work is funded in part by National Science Foundation under grant number 1516636 and 1723765.

References

[1] Eric Y, Esfahani N, Malek S. A systematic survey of self-protecting software systems. ACM Trans Auton Adapt Syst (TAAS) 2014;8.4:17.
[2] Mazeiar S, Pasquale L, Omoronyia I, Ali R, Nuseibeh B. Requirements-driven adaptive security: protecting variable assets at runtime. Requirements engineering conference (RE), 2012 20th IEEE international. IEEE; 2012.
[3] Christopher B, Montrieux L, De Lemos R, Yu Y, Wermelinger M. Run-time generation, transformation, and verification of access control models for self-protection. In: Proceedings of the 9th international symposium on software engineering for adaptive and self-managing systems; 2014. p. 135–44. ACM.
[4] Srdjan M, Dulay N, Sloman M. Rumpole: an introspective break-glass access control language. ACM Trans Inf Syst Secur(TISSEC) 2014;17(1):2.
[5] Moitrayee C, Namin AS. Detecting web spams using evidence theory. In: 2018 IEEE 42nd annual computer software and applications conference (COMPSAC); 2018. p. 695–700. IEEE.
[6] Sara S, Namin AS. Adaptive reasoning for context-sensitive access controls. Computer software and applications conference (COMPSAC), 2016 IEEE 40th annual. Vol. 1. IEEE; 2016.
[7] Lorenzo B, Bertino E, Hussain SR. A system for profiling and monitoring database access patterns by application programs for anomaly detection. IEEE Trans Software Eng 2017;43.5:415–31.
[8] Sara S, Namin AS. Poster: reasoning based on imperfect context data in adaptive security. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol. 2; 2015. p. 835–6. IEEE.
[9] Sandhu Ravi S, Coynek EJ, Feinsteink HL, Youmank CE. Role-based access control models. IEEE Comput 1996;29(2):38–47.
[10] El KAA, Baida REI, Balbiani P, Benferhat S, Cuppens F, Deswarte Y, Miege A, Saurel C, Trouessin G. Organization based access control. In: Policies for distributed systems and networks, 2003. Proceedings. POLICY 2003. IEEE 4th international workshop on; 2003. p. 120–31. IEEE.
[11] Fabien A, Cuppens F, Cuppens-Boulahia N, Coma C. MotorBAC 2: a security policy tool. In: 3rd conference on security in network architectures and information systems (SAR-SSI 2008). France: Loctudy; 2008. p. 273–88.
[12] Frederic C, Cuppens-Boulahia N, Ghorbel MB. High level conflict management strategies in advanced access control models. Electron Notes Theor Comput Sci 186(2007):3–26.

[13] Gerhard B, Eiter T, Truszczynski M. Answer set programming at a glance. Commun ACM 2011;54(12):92–103.
[14] Vladimir L. What is answer set programming? AAAI 2008;8:1594–7.
[15] Raymond R. A logic for default reasoning. Artif Intell 1980;13(1):81–132.
[16] Michael G, Lifschitz V. The stable model semantics for logic programming. ICLP/SLP 1988;88:1070–80.
[17] Martin G, Kaufmann B, Neumann A, Schaub T. Clasp: a conflict-driven answer set solver. In: International conference on logic programming and nonmonotonic reasoning. Springer Berlin Heidelberg; 2007. p. 260–5.
[18] Nicola L, Pfeifer G, Faber W, Eiter T, Gottlob G, Perri S, Scarcello F. The DLV system for knowledge representation and reasoning. ACM Trans Comput Log (TOCL) 2006;7(3):499–562.
[19] Martin G, Schaub T, Thiele S. Gringo: a new grounder for answer set programming. In: International conference on logic programming and nonmonotonic reasoning. Springer Berlin Heidelberg; 2007. p. 266–71.
[20] Gelfond M, Kahl Y. Knowledge representation, reasoning, and the design of intelligent agents: the answer-set programming approach. Cambridge, England: Cambridge University Press; 2014.
[21] Baral C. Knowledge representation, reasoning and declarative problem solving. Cambridge university press; 2003.
[22] Erdem E, Gelfond M, Leone N. Applications of answer set programming. AI Mag 2016;37.3.
[23] Ahn G-J, Hu H, Lee J, Meng Y. Representing and reasoning about web access control policies. In: Computer software and applications conference (COMPSAC), 2010 IEEE 34th Annual; 2010. p. 137–46. IEEE.
[24] Ayed D, Lepareux M-N, Martins C. Analysis of XACML policies with ASP. New Technol Mobil Secur (NTMS), 2015 7th Int Conf IEEE 2015.
[25] Hu H, Ahn G-J, Jorgensen J. Multiparty access control for online social networks: model and mechanisms. IEEE Trans Knowl Data Eng 2013;25(7):1614–27.
[26] Gelfond M, Lobo J. Authorization and obligation policies in dynamic systems. In: In International conference on logic programming. Berlin, Heidelberg: Springer; 2008. p. 22–36.
[27] Kuhn DR, Coyne EJ, Weil TR. Adding attributes to role-based access control. Computer (Long Beach Calif) 2010;43.6:79–81.
[28] Moffett J, Sloman M, Twidle K. Specifying discretionary access control policy for distributed systems. Comput Commun 1990;13(9):571–80.
[29] Bell DE, Padula LJL. Secure computer system: Unified exposition and multics interpretation. No. MTR-2997-REV-1. Bedford MA: Mitre Corp; 1990.
[30] Thomas RK, Sandhu RS. Task-based authorization controls (TBAC): a family of models for active and enterprise-oriented authorization management. In: Database security XI. US: Springer; 1998. p. 166–81.
[31] Bertino E, Bonatti PA, Ferrari E. TRBAC: A temporal role-based access control model. ACM Trans Inf Syst Secur(TISSEC) 2001;4(3):191–233.
[32] Karp A, Haury H, Davis M. From ABAC to ZBAC: the evolution of access control models. International conference on cyber warfare and security. Academic conferences international limited; 2010.
[33] Godik S, Moses T. OASIS Extensible access control markup language (XACML). OASIS committee secification cs-xacml-specification-1.0; 2002. Harvard.
[34] Hughes G, Bultan T. Automated verification of access control policies using a sat solver. Int J Softwa Tool Technol Transf (STTT) 2008;10.6:503–20.
[35] Turkmen F, Hartog Jd, Ranise S, Zannone N. Formal analysis of XACML policies using SMT. Comput Secur 66(2017):185–203.
[36] Fisler K, Krishnamurthi S, Meyerovich LA, Tschantz MC. Verification and change-impact analysis of access-control policies. In: Proceedings of the 27th international conference on software engineering. ACM; 2005.
[37] Stepien B, Felty A, Matwin S. Challenges of composing XACML policies. Availability, reliability and security (ARES), 2014 ninth international conference on. IEEE; 2014.
[38] Turkmen F, Hartog Jd, Ranise S, Zannone N. Analysis of XACML policies with SMT. International conference on principles of security and trust. Berlin, Heidelberg: Springer; 2015.
[39] Turkmen F, Hartog Jd, Zannone N. POSTER: Analyzing access control policies with SMT. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. ACM; 2014.
[40] Crampton J, Morisset C, Zannone N. On missing attributes in access control: non-deterministic and probabilistic attribute retrieval. Proceedings of the 20th ACM Symposium on Access Control Models and Technologies ACM 2015.
[41] Sartoli S, Namin AS. A semantic model for action-based adaptive security. In: Proceedings of the symposium on applied computing. ACM; 2017.
[42] Bettini C, Brdiczka O, Henricksen K, Indulska J, Nicklas D, Ranganathan A, Riboni D. A survey of context modelling and reasoning techniques. Pervasive Mob Comput 2010;6(2):161–80.
[43] Perera C, Zaslavsky A, Christen P, Georgakopoulos D. Context aware computing for the internet of things: a survey. IEEE Commun Surve Tutor 2014;16(1):414–54.
[44] Ali R, Dalpiaz a, Giorgini P. A goal-based framework for contextual requirements modeling and analysis. Requir Eng 2010;15(4):439–58.
[45] Shaikh RA, Adi K, Logrippo L. A data classification method for inconsistency and incompleteness detection in access control policy sets. Int J Inf Secur 2016:1–23.
[46] Rao P, Lin D, Bertino E, Li N, Lobo J. An algebra for fine-grained integration of XACML policies. In: Proceedings of the 14th ACM symposium on access control models and technologies; 2009. p. 63–72. ACM.
[47] Crampton J, Morisset C. PTACL: a language for attribute-based access control in open systems. In: International conference on principles of security and trust. Berlin Heidelberg: Springer; 2012. p. 390–409.
[48] Jajodia S, Samarati P, Sapino ML, Subrahmanian VS. Flexible support for multiple access control policies. ACM Trans Database Syst (TODS) 2001;26.2:214–60.
[49] Bauer L, Garriss S, Reiter MK. Detecting and resolving policy misconfigurations in access-control systems. ACM Trans Inf Syst Secur(TISSEC) 2011;14.1:2.
[50] Baracaldo N, Joshi J. An adaptive risk management and access control framework to mitigate insider threats. Comput Secur 2013;39:237–54.
[51] Basin D, Burri SJ, Karjoth G. Obstruction-free authorization enforcement: aligning security and business objectives. J Comput Secur 2014;22(5):661–98.
[52] Balai E, Gelfond M, Zhang Y. Towards answer set programming with sorts. In: International conference on logic programming and nonmonotonic reasoning. Berlin Heidelberg: Springer; 2013. p. 135–47.
[53] Autrel F. MotorBAC. (march 2017). retrieved march 17, 2017. 2017. https://sourceforge.net/projects/motorbac/ IEEE, 2006.
[54] Tsigkanos C, Pasquale L, Menghi C, Ghezzi C, Nuseibeh B. Engineering topology aware adaptive security: preventing requirements violations at runtime. In: Requirements engineering conference (RE). IEEE; 2014. p. 203–12.
[55] Kevin T, Dulay N, Lupu E, Sloman M. Ponder2: a policy system for autonomous pervasive environments. In: Autonomic and autonomous systems, 2009. ICAS'09. Fifth International Conference on. IEEE; 2009.
[56] Elda P, Dalpiaz F, Poggianella M, Roberti P, Giorgini P. STS-Tool: socio-technical security requirements through social commitments. Requirements engineering conference (RE), 2012 20th IEEE international. IEEE; 2012.
[57] https://www.axiomatics.com/product/developer-tools-and-apis/.
[58] Jianjun Z, Namin AS. A markov decision process to determine optimal policies in moving target. The 25th ACM conference on computer and communications security (ACM CCS). ACM; 2018.