# A Grammar for Specifying Usage Control Policies

Rafael Teigão, Carlos Maziero, Altair Santin
Graduate Program in Computer Science
Pontifical Catholic University of Paraná
80215–901 Curitiba, Brazil
Email: {teigao,maziero,santin}@ppgia.pucpr.br

***Abstract*—Usage control goes beyond traditional access control, addressing its limitations related to attribute mutability and continuous usage permission validation. The recently proposed UCON$_{\mathrm{ABC}}$ model establishes an underlying mathematical framework to deal with the new needs of security and control systems. That model was only described by a logic specification, and this paper proposes implementing it as an LALR(1) grammar, which is defined here. The proposed grammar is then used for representing common access and usage control scenarios, showing its expressiveness and usefulness. The proposed grammar is being incorporated into a file usage control mechanism implemented on a COTS operating system.**

## I. Introduction

Classic access control mechanisms are not suitable for the vast amount of ways data (content) is available today. For instance, *Digital Rights Management* (DRM) introduces the necessity for control that goes beyond the simple one-step access granting. That is also true when data are collected from several independent sources, such as medical information from patients of an hospital. Electronic commerce of digital items, nowadays, brings with it the necessity of checking whether some additional requirements have been met, like accepting an End-User License Agreement (EULA) and so on.

Usage control (UCON), introduced by Park and Sandhu [1] defines on-going checks of attributes/requirements (e.g. permissions of a user to watch a movie) and consider the mutability of that attributes during usage. Furthermore, the authors presented models based on external information (not considered in classic access control models) that is taken into account in the model. The UCON$_{\mathrm{ABC}}$ model [2] is based on the concepts of *Authorization*, *oBligations*, and *Conditions*, supporting the current control needs while still supporting classic access control. Access and usage controls are based on rules that specify requirements to get access or usage rights over objects, respectively. The set of rules defines a policy, which is written in a policy language.

Based on the UCON$_{\mathrm{ABC}}$ logical specification [3], this paper proposes a grammar for specifying UCON policies, which has been conceived to be expressive enough in order to represent authorization, obligation and condition statements. The aim of this grammar is to allow a sysadmin to associate attributes to users and requirements to objects, and to impose rules that use these attributes and requirements to make access and usage decisions. The grammar definition should avoid ambiguities: each statement must have a single, unambiguous, and clear semantics.

This paper is organized as follows. Section II brings an overview of the UCON$_{\mathrm{ABC}}$ model; section III details the proposed UCON$_{\mathrm{ABC}}$ grammar; section IV gives some details of the implementation prototype; examples of the grammar representing common usage and access control scenarios are given in section V; section VI presents some related work; finally, section VII summarizes this paper and present future directions for this work.

## II. The UCON$_{\mathrm{ABC}}$ usage control model

Every current access control mechanism relates user's and object's attributes to make access decisions, but these attributes are usually very specific to the mechanism they pertain to and in most cases can only be changed by administrative actions. UCON$_{\mathrm{ABC}}$ [1], [2] introduces attribute mutability, which brings with it the possibility to influence current or future actions based upon usage history, for example. It is possible to correlate usage decisions based on attributes that are altered on each access (take a credit based system for example: with each access, the user credit is decremented by some value and, when the user is out of credit, her access can be denied). Beyond mutability, attributes are also mechanism independent: Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role-Based Access Control (RBAC), among many others, can be implemented within UCON$_{\mathrm{ABC}}$, without changing attribute properties.

Authorization is only one aspect of UCON$_{\mathrm{ABC}}$ that can be evaluated when making decisions. Obligations and conditions may also be evaluated. While authorization encompasses traditional access controls, obligations permit verifying mandatory actions a user has to perform before or during usage. Conditions on the other hand are environmental requirements that can be taken into account.

Rule evaluation can be performed before (*pre*) or during (*on*) usage/access[1]. Attribute updates can be done before (1), during (2) or after (3) usage, or they can never be updated (0). For instance, if one were to denote an authorization (A) process that is going to take place during usage (*on*), but will only update attributes after usage (3), one could write UCON$_{\mathrm{onA_3}}$. Table I shows the UCON$_{\mathrm{onA_3}}$ model space. It is important to notice that, since conditions are environmental

[1]Although authorizations and conditions usually relate to access control and obligations to usage control, we avoid differentiating them, for the sake of simplicity.

information (i.e. the system checking them have no influence on their values), there are no attribute updates related to them.

TABLE I
THE 16 BASIC UCON$_{ABC}$ MODELS [2]

| | 0<br>immutable | 1<br>pre-update | 2<br>ongoing-update | 3<br>post-update |
|---|---|---|---|---|
| preA | Y | Y | N | Y |
| onA | Y | Y | Y | Y |
| preB | Y | Y | N | Y |
| onB | Y | Y | Y | Y |
| preC | Y | N | N | N |
| onC | Y | N | N | N |

## III. THE PROPOSED GRAMMAR

The objective of the grammar proposed here is to be as close as possible to the UCON formal description, without loosing the ability of being implemented in a real system. Also, the grammar should be kept simple, in order to allow an efficient (fast) implementation. Grammars may be described in a format known as the Extended Backus-Naur-Form (EBNF) [4] and they are composed of terminal and non-terminal symbols. We are not concerned with the non-terminal symbols in this work because they are of little help for understanding the grammar. In addition, the complete formal description of the grammar is too long to fit in this paper. So, it will be presented here only informally.

### *A. Language requirements*

For the description of control rules to be secure and efficient, it has to follow some criteria, as detailed bellow:

- The language defined by the grammar should have no ambiguities, and should clearly express the functional predicates for authorizations, obligations, and conditions. Ambiguous grammars have bigger processing and memory requirements, because more tokens should be looked-ahead in order to solve ambiguities.
- The access should only be granted after all evaluated rules returned `true`. Each rule expressed will have the ability to altogether deny a usage request if it returns `false`, but if there is not at least a single rule that explicitly denies usage, the request should be granted. Therefore, an empty rule-set does not have the power to deny access. This approach was also used in [5].
- It should be possible to add new control rules without modifying user's attributes, provided that the required attributes are already present.
- The evaluation process must be efficient, otherwise the time required for making a decision may hinder the user experience and degrade the system performance (i.e. it should not take much longer than traditional access control mechanisms to grant or deny access).
- The proposal should be fully implementable and should be easy to integrate into existing environments.

### *B. Representing attributes, obligations, and conditions*

It is important to observe that the parser for this grammar is meant to be incorporated into a real system. Therefore it must be practical and usable. This requirement introduces some limitations such as controls to assure the integrity of a policy file, for example. Other limitations are inherent to the translation from the formal (mathematical) model to the real implementation. The way we represent attributes, obligations and conditions are clear examples of these limitations. Attributes are implemented as variables which may have two different types: `integer` and `string`.

An Attribute variable is represented by a $ followed by its name (e.g. `$name`). When a variable is created, a value must be assigned to it. The value initially assigned to a symbol define its type: if a symbol is first assigned as an integer value, then it will always be treated as an integer.

Obligations and conditions have special needs regarding the data that may be hold by the structures representing them. As obligations and conditions relate to external information (outside the policy enforcer), there is a security issue to be considered, in order to avoid user-supplied code to be executed during their evaluation. Since one cannot allow a code supplied by the user to be run at the same security level as the enforcer, we limit the amount and kind of data that can be used. Obligations are presented as slots, data holders that can be filled by an external program, and that can be read by the enforcer. Such slots can be accessed by using the keyword `o$slot` followed by the slot number (e.g. `o$slot 37`).

Similarly, a keyword is used to access information on conditions. Only predefined conditions are available, in the current prototype they are the time of day (`c$time`), amount of CPU used (`c$cpu_used`), amount of free memory (`c$free_mem`), and partition space (`c$free_disk`). These specific conditions where selected to be implemented because they relate strongly with the current prototype environment, an operating system (cf. section IV). However, conditions can easily be extended by adding new constructs to the grammar relating a keyword to a function that returns the required information.

### *C. Terminal Symbols*

Grammar's terminal symbols consist of variable names, keywords, constant values, and operators. As it will be shown in session III-B, the representation of some model aspects is met with limitations, specially related to obligations and conditions. The following tables present the terminal symbols: table II shows those symbols related to variables and values, and table III introduces the operators and their functionality (the operators are shown in increasing precedence order).

Some operators may be used in sequence. For instance, it is possible to test whether the value stored in a variable is greater than the difference of two variables: `$credit` > `$cost` − `$discount`.

TABLE II
VARIABLES AND VALUES SYMBOLS

| Symbol | Type | Description |
|---|---|---|
| $*name* | integer and string | A named variable, beginning with the $ symbol, followed by its distinguishing name. |
| *digits* | integer | A constant integer value. |
| $[string_1, \ldots, string_n]$ | string | A constant set of strings. |
| o$slot | integer | Keyword to access obligation values. |
| c$*name* | integer | Keyword to access a value for the resource *name*, used to express conditions. |

TABLE III
OPERATOR SYMBOLS

| Operator | Type | Functionality |
|---|---|---|
| `=` | Assignment | Assigns the value on the right to the variable on the left. |
| `== != < > <= >=` | Comparison | Perform comparisons between left and right operands. |
| `& \|` | Logical | Logical operators AND and OR. |
| `size` | Set operation | Returns the number of elements in a set. |
| `+ - * /` | Arithmetic | Arithmetic operations between left and right numeric operands. |
| `+ *` | Set operation | Union and intersection between left and right set operands. |
| (*expression*) | Precedence | The inner-most expression should be analyzed before the non-parenthesized expression. |
| `#` | - | Starts a comment. |

## IV. THE PROTOTYPE IMPLEMENTATION

The approach proposed in this paper is being implemented as a proof-of-concept system, divided into a rule parser, a reference monitor and a policy enforcer. The parser translates the rules expressed by the grammar into an internal representation to be used by the reference monitor. The process of evaluating a policy consists on setting the semantic values to the rules (associating requirements to objects) expressed in the grammar, and deciding if the subject attributes meets the objects' required rights. If so, the request is allowed, otherwise denied. If any rule fails, the reference monitor returns `false` to the enforcer, which must take the appropriate actions to revoke the active permissions (if there are any) and to prevent the access or use of the given object.

The grammar proposed here was partially implemented as a proof-of-concept prototype, which is being incorporated in a Linux kernel, in order to allow a sysadmin to define usage policies for file resources. The enforcement module intercepts system calls used in file operations [6], such as `open`, `close`, `read`, and `write`. The *pre* checks and attributes updating are performed during `open` requests, while on-going actions are performed in the `read` and `write` calls. *Pos* updates are realized when the `close` is called, or when a previous checks results in access denied. As the file usage is achieved through the `read` and `write` operations[2], mutable attributes can be checked again before such operations are performed.

[2]For the sake of simplicity, we are not yet considering other file-related operations, like `lseek`, `mmap`, and others.

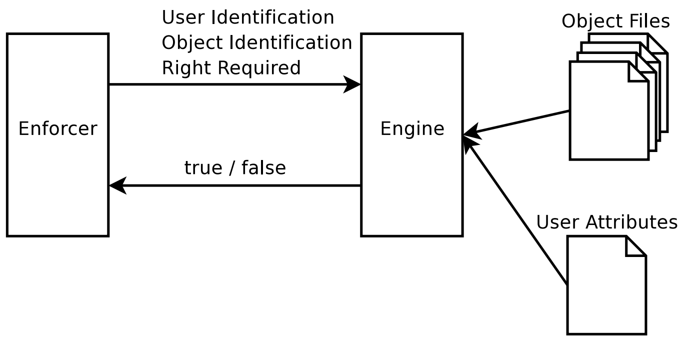

Fig. 1. Prototype overview

This approach is consistent with the model presented in [3].

The enforcer is the entry point of the system. Each relevant system call will trigger the enforcer, which will set the right requested and consult the reference monitor for a decision. The monitor will then evaluate each rule and the set of rights, in order to decide granting or denying usage or access. When the enforcer receives an answer from the monitor, it will take the appropriate actions to allow or deny the access. Figure 1 illustrates this process.

The prototype is being implemented in C, using Bison [7] for defining a LALR(1) grammar [8]. The implementation has about 500 lines of C code, and the rule parser has 35 constructs and 60 states. Each one of the requirements stated on section III-A is being covered in our proposal:

- Ambiguity is avoided by assuring that the defined grammar is LALR(1);
- Rules can be evaluated efficiently, by using few constructs to define the grammar.
- By separating the rules and attributes into different files, it is easy to add a new rule, simply by editing a single file. It is not necessary to fuss with user's attributes to create a new rule, when all the required attributes are already present.
- Since the semantic associated with a grammar construct returns immediately a `false` value every time an evaluated rule fails, any rule has the ability to altogether deny a usage request, independently of the associated semantic value of the other rules.

The next section will show some examples of policies that can be described by the grammar defined here, in order to show is expressiveness.

## V. USING THE GRAMMAR

Policies are defined as combinations of attributes, integer and string constants, obligations, conditions, and the operators related to them. Each object has an attribute file and three policy files associated to them (see Fig. 2):

- The attribute file holds the attributes initialization and should be read before the other files.
- The first policy file contains the *pre* policies and the *pre* updates.
- The second file holds the *on* policies and updates.

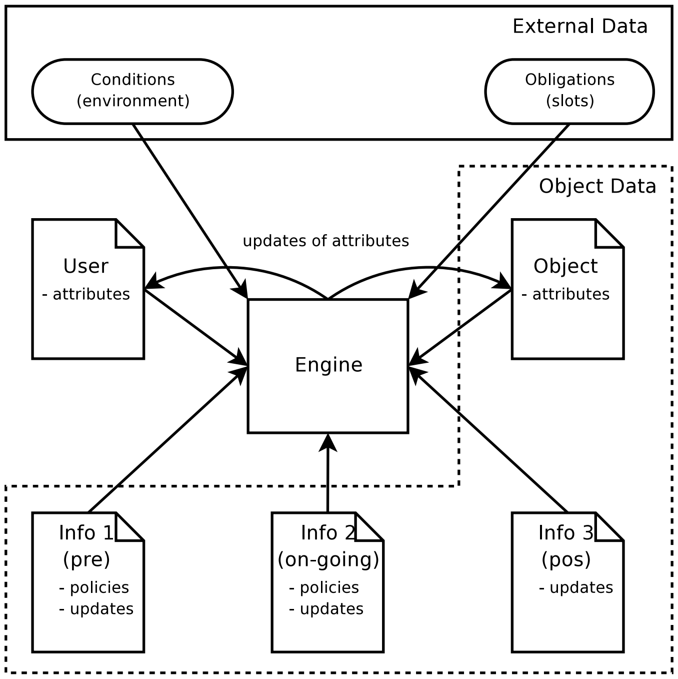

Fig. 2. Files used to store the policies

- The third file holds the *pos* updates. Note that there are no *pos* policies in the original model, but this grammar does not enforce their exclusion from the third file.

Each user has only one file containing the user's attributes, and does not have policies directly associated to her. An authorization policy could be as simple as:

```
$owner_id = 7503     # owner's user ID (UID)
$usr_id == $owner_id # UID and owner's ID must match
```

These statements are to be placed within the object's first file (*pre*), meaning that the access will be granted if the user's id matches the id of the object's owner. Each one of the three rule files are read and evaluated in sequence. If any rule returns `false`, then the user has its usage request denied and the remaining of the file is not evaluated. If the file being evaluated describes a *pre* or *on* policy, then at least the file containing the *pos* updates is also evaluated, assuring that the attributes are correctly updated.

In this section we will describe some examples of policies defined using the language described by the proposed grammar. For each example, the *pre*, *on*, and *pos* updates and policies will be separated as necessary into the three different files used for this purpose. It is interesting to contrast some of the examples shown here against those on [2], in order to evaluate the implementation compatibility with the formal model.

### A. DAC with ACL, using $UCON_{\mathrm{preA}_0}$

Discretionary Access Control [9] with Access Control Lists is a very simple and straightforward control mechanism, which only takes a few lines to be implemented within this grammar. The user's attributes should only contain her identification: `$usr_id = 5456`. The enforcer will pass the right requested as a integer, 0 for read and 1 for write access: `$right = 0`. The object's attributes file should look like:

```
# set of all UIDs with read permission
$obj_read_perm = [1549 4334 5456 8997]

# set of all UIDs with write permission
$obj_write_perm = [4456 5456 7896 8345]
```

These attributes are groups of users' IDs that have read or write permission. The *pre* policies file could simply contain:

```
($right == 0 &  # attempting read access
  # the UID must belong to $obj_read_perm
  (size ($usr_id * $obj_read_perm) != 0)
) |
($right == 1 &  # attempting write access
  # the UID must belong to $obj_write_perm
  (size ($usr_id * $obj_write_perm) != 0) )
```

This policy basically states that if a user is requesting read or write permission, the user's ID should be present in the `$obj_read_perm` or `$obj_write_perm` list, respectively; otherwise, the access is denied.

### B. MAC policies, using $UCON_{\mathrm{preA}_0}$

Mandatory Access Control [10] may be implemented by considering clearance as an user's attribute, and classification as an object's attribute. The user's file could simply contain the clearance level: `$clearance = 5`. Likewise, the object's attribute file could simply contain its classification level: `$classification = 3`. To ascertain that to grant read access, clearance must be greater or equal to classification, and to grant write access, clearance must be lesser or equal to classification, the *pre* policies file should state:

```
# don't read up
($right == 0 & ($clearance >= $classification) )
|
# don't write down
($right == 1 & ($clearance <= $classification) )
```

This assumes that the enforcer fills the variable indicating the right required (`$right`), as it was done in the previous example.

### C. The user has to watch advertisements while exercising a right, using $UCON_{\mathrm{onB}_0}$

The user has to keep an advertisement window open all the time a certain right over an object is exercised. An external program, possibly the one controlling the window, will update an obligation slot indexed by the user's ID. The user's attribute file contains only this ID: `$usr_id = 5899`. The object has only one file, containing the *on* policies, which is going to be checked every time the right over the object is to be performed (e.g. when reading the next seconds from a music file):

```
# access to the slot indexed by UID
o$slot $usr_id == 1
```

The slot indexed by the user's ID is created by the system when the user requests access to the object. The external program that controls the advertisement window writes 1 in this slot once the window is opened, and changes it to 0 when it is closed.

### *D. Simultaneous accesses limits, controlled by time schedules, using UCON*$_{\mathrm{preA}_{13}\mathrm{preC}_0}$

A given object can be accessed at most by 10 simultaneous users between 8am and 6pm and by 20 users after 6pm and before 8am. Already accessing users will not have their access revoked when the time shifts for a period admitting a smaller number of simultaneous access, but no new user will be accepted until their number drops and a vacancy is created. The object's attributes file contains the number of simultaneous accesses accepted and the start and end times for the reduction on the number of users:

```
$users = 0        # number of current users
$max_day = 10     # max users during the day
$max_night = 20   # max users during the night
$day_start = 8    # day starts 8 o'clock (8am)
$day_end = 18     # day ends 18 o'clock (6pm)
```

The *pre* policies file controls the number of simultaneous users and updates the variable controlling this number:

```
(
  ( # it's day period
    (c$time >= $day_start) & (c$time <= $day_end) )
  & ($users < $max_day) )
|
(
  ( # it's night period
    (c$time < $day_start) & (c$time > $day_end) )
  & ($users < $max_night) )

# increments current users total
$users = $users + 1
```

Note that at the last line, if the user is granted access, then the number of users is updated. If the first rule fails, the parsing of the file stops and the update is not performed. There should also be a *pos* update file, for decreasing the number of current users when an user stops accessing the object:

```
# decreases current users total
$users = $users - 1
```

### *E. Limited number of simultaneous usages, revocation using usage time, using UCON*$_{onA_{123}}$

The user's attribute file will have a variable to track its total usage time and a variable to record the time of its last action:

```
$total_usage = 0 # total usage time (in hours)
$last_action = 0 # time of last action
```

The object should also have a variable to set the maximal number of users it accepts, another variable to set the limit of usage time (6 hours in this case) and a third one to track the amount of simultaneous users:

```
$max_users = 10 # max number of users allowed
$max_usage = 6  # max number of hours per user
$users = 0      # number of current users
```

Now, the policies will have to be divided into three files. The *pre* files will be:

```
$users < $max_users
$users = $users + 1

# stores the time this action was performed
$last_action = c$time
```

The *on* file will be:

```
# increments $total_usage by the amount of
# time between this and the last action
$total_usage = $total_usage + (c$time-$last_action)

# $max_usage must not be hit
$max_usage > $total_usage
$last_action = c$time
```

Finally, the *pos* file will be:

```
# when the access ends, the variables must be reset
$total_usage = 0
$last_action = 0
$users = $users - 1
```

By zeroing the user's attributes, she is able to release the object and request it again, giving chance to others to have their access granted without loosing the object if there are no other users on the queue. This policy could be modified to leave the total usage recorded, requiring an administrator's action to allow the user to access the object again.

### *F. Flat RBAC*

Three of the requirements for Flat RBAC (Role-Based Access Control, level 1) [11] is part of the user's attributes:

```
# roles available to this user
$roles = [director manager teller]

# set of currently active roles
$active_roles = [manager teller]
```

These two variables provide many-to-many user-role assignment ($roles is a group of roles, and each role can be assigned to other users), support for user-role assignment review (one can ask for the contents of $role) and a user can activate multiple roles simultaneously ($active_roles is also a group of roles). The actual access control is implemented as a combination of object's attributes and policies. A sole object's attribute is required:

```
$required_roles = [teller manager]
```

The *pre* policies should take care that at least one of the required roles should be present on the user's roles list, and update the active roles to add this role:

```
# object's required roles and roles available
# to the user must have a non-null intersection
size ($required_roles * $roles) != 0

# mark the role as active
$active_role = $active_role +
               ($required_roles * $roles)
```

### *G. Constrained RBAC*

Now we add what is necessary to reach RBAC level 3, also known as Constrained RBAC: Separation of Duty (SoD) support. This is basically a modification on *pre* and the addition of *on* policies. The *pre* policies now should ascertain that two conflicting roles cannot be activated at the same time:

```
size ($active_role * $required_roles) == 1
```

We are assuming that the roles listed on `$required_roles` are mutually exclusive. If that were not the case, we could simply add another variable to the object's attributes, listing conflicting roles. The following *on* policy guarantee that, if a conflicting role is activated by an access to another object (in which these roles are non-conflicting ones) or if the required role is no longer active, then this access will be revoked:

```
size($active_role * $required_roles) == 1
```

This rule is just the same added to the *pre* file, but since it is verified during the access, it should be placed on its own file.

## VI. Related Work

There are several languages proposed to address different security and privacy needs. The *Enterprise Privacy Authorization Language* (EPAL) [12] tries to unify the rules that control how privacy sensitive information should be handled across systems, by creating a universal mechanism for describing required privacy policies.

In [13], the authors propose a modular authorization language to support distributed authorization between cooperating administrative teams, based mostly on RBAC. Woo and Lam [5] also tackle distributed authorization by introducing a language to encode authorization requirements, which they call a *Policy Base*; on another paper [14] they further introduce the formal syntax and semantics for a language called *Generalized Access Control List* (GACL) for representing authorization policies based upon ACL.

Since GACL is limited to ACL-based mechanisms, Ryutov and Neuman [15] present a policy language that allows to represent several control models (such as ACL, capabilities, lattice-based and RBAC) and a generic authorization and access-control API (GAA API) to facilitate integration of authentication and authorization.

Finally, there are two application-specific implementations of $\text{UCON}_{\text{ABC}}$. The first is the application of the model to B2B systems [16] and the second is an extension to CORBASec to integrate some $\text{UCON}_{\text{ABC}}$ features [17].

Just like the $\text{UCON}_{\text{ABC}}$ model is distinct from traditional models by supporting modern needs of continuous usage control and privacy, such as credit-based usage and DRM, the language represented by our grammar differs from the above presented languages by including means to express a more abstract view of control.

## VII. Conclusion

This paper presented a LALR(1) grammar for specifying UCON policies. The proposed grammar is simple to understand but expressive enough to describe a wide range of policies, like DAC, MAC, RBAC, UCON, and DRM-like policies, as shown in section sec:policies.

Apart from the limitations related to the representation of obligations and conditions, it is not hard to bring the power of $\text{UCON}_{\text{ABC}}$ to a real system. There are other examples [16], [17] of $\text{UCON}_{\text{ABC}}$ implementations, but they are not so complete and expressive as the work presented here.

We are now working to integrate the grammar and its enforcer presented in this paper into a real operating system, and also refining the grammar in the process as well. There are still open issues to be solved, such as efficiently relating the rules and attributes files to objects and creating and managing obligation slots, which could be overlooked in our proof-of-concept implementation, but are essential for a full functional system.

We believe that having a grammar, such as the one presented here, and an efficient enforcer, can ease the adoption of $\text{UCON}_{\text{ABC}}$ as a powerful access control, DRM, and data protection solution.

## References

[1] J. Park and R. Sandhu, "Usage control: A vision for next generation access control." in 2nd *International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, ser. Lecture Notes in Computer Science, V. Gorodetsky, L. J. Popyack, and V. A. Skormin, Eds., vol. 2776. Springer, 2003, pp. 17–31.

[2] ——, "The $UCON_{ABC}$ usage control model," *ACM Transactions on Information and System Security*, vol. 7, no. 1, pp. 128–174, 2004.

[3] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu, "A logical specification for usage control," in *SACMAT '04: Proceedings of the 9th ACM symposium on Access control models and technologies*. New York, NY, USA: ACM Press, 2004, pp. 1–10.

[4] International Organization for Standardization, *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. Genève, Switzerland: International Organization for Standardization, 1996.

[5] T. Y. C. Woo and S. S. Lam, "Authorization in distributed systems: a formal approach," in *IEEE Symposium on Research in Security and Privacy*, 1992, pp. 33–51.

[6] T. Garfinkel, "Traps and pitfalls: Practical problems in in system call interposition based security tools," in *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[7] R. Stallman and C. Donnelly, *Bison – The Yacc-compatible Parser Generator*. Free Software Foundation, Inc., 2005.

[8] F. DeRemer and T. Pennello, "Efficient computation of LALR(1) look-ahead sets," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 615–649, 1982.

[9] B. W. Lampson, "Protection," *SIGOPS Operating System Review*, vol. 8, no. 1, pp. 18–24, 1974.

[10] D. Bell and L. LaPadula, "Secure computer systems: Unified exposition and multics interpretation," MITRE Corporation, Massachusetts, USA, Tech. Rep., March 1976.

[11] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The NIST model for role-based access control: towards a unified standard," in *RBAC '00: Proceedings of the fifth ACM workshop on Role-based access control*. New York, NY, USA: ACM Press, 2000, pp. 47–63.

[12] M. J. May, "Privacy system encoded using EPAL 1.2," University of Pennsylvania, Tech. Rep., August 2004.

[13] H. F. Wedde and M. Lischka, "Modular authorization and administration," *ACM Transactions on Information and System Security*, vol. 7, no. 3, pp. 363–391, 2004.

[14] T. Y. C. Woo and S. S. Lam, "Designing a distributed authorization service," in *IEEE INFOCOM*, 1998, pp. 419–429.

[15] T. Ryutov and C. Neuman, "Representation and evaluation of security policies for distributed system services," in *DARPA Information Survivability Conference Exposition*, Healton Head, South Carolina, January 2000.

[16] A. Camy, C. M. Westphall, and R. Righi, "Aplicação do modelo $UCON_{ABC}$ em sistemas de comércio eletrônico B2B," in 5th *Brazilian Symposium on Information Security and Computing Systems (SBSeg)*. Brazilian Computing Society (SBC), September 2005, in Portuguese.

[17] M. S. Higashiyama, L. C. Lung, R. Obelheiro, and J. da Silva Fraga, "JaCoWeb-ABC: Integração do modelo de controle de acesso $UCON_{ABC}$ no CORBASec," in 5th *Brazilian Symposium on Information Security and Computing Systems (SBSeg)*. Brazilian Computing Society (SBC), September 2005, in Portuguese.